KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Fuzzing Intel SGX Enclaves

Thomas De Backer

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen, hoofdoptie
Gedistribueerde systemen

Promotoren:
Dr. J.T. Mühlberg
Prof. dr. ir. F. Piessens

Assessoren:
Dr. A. Akkasi
Dr. R. Strackx

Begeleiders:
Dr. J.T. Mühlberg
ir. J. Van Bulck

Academiejaar 2018 – 2019

© Copyright KU Leuven

Without written permission of the thesis supervisors and the author it is forbidden to reproduce or adapt in any form or by any means any part of this publication. Requests for obtaining the right to reproduce or utilize parts of this publication should be addressed to the Departement Computerwetenschappen, Celestijnenlaan 200A bus 2402, B-3001 Heverlee, +32-16-327700 or by email info@cs.kuleuven.be.

A written permission of the thesis supervisors is also required to use the methods, products, schematics and programmes described in this work for industrial or commercial use, and for submitting this publication in scientific contests.

Zonder voorafgaande schriftelijke toestemming van zowel de promotoren als de auteur is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen tot of informatie i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, wend u tot het Departement Computerwetenschappen, Celestijnenlaan 200A bus 2402, B-3001 Heverlee, +32-16-327700 of via e-mail info@cs.kuleuven.be.

Voorafgaande schriftelijke toestemming van de promotoren is eveneens vereist voor het aanwenden van de in deze masterproef beschreven (originele) methoden, producten, schakelingen en programma's voor industrieel of commercieel nut en voor de inzending van deze publicatie ter deelname aan wetenschappelijke prijzen of wedstrijden.

Preface

First and foremost, I would like to thank my supervisors for their valuable input, patience and feedback. This work would not have gotten finished if it were not for them. I would also like to thank my friends and family for their unending support and reminding me not to procrastinate too much. It was fun working on this thesis, and I have learned a lot about the subject and myself while doing so.

Thomas De Backer

Contents

Preface i

Abstract iv

Samenvatting v

List of Figures vii

1 Introduction 1
1.1 Motivation 1
1.2 Goals of this thesis 1
1.3 Overview 2

2 Background and related work 5
2.1 Operating System concepts 5
2.2 Software security 8
2.3 Intel SGX concepts 9
2.4 Previous work 13
2.5 Conclusion 20

3 Fuzzing Intel SGX enclave execution 21
3.1 Problem statement 21
3.2 Fuzzing concepts 22

4 Developing a Bug Oracle 27
4.1 Goal of this chapter 27
4.2 Development process of the bug oracle 31
4.3 Conclusion 38

5 Evaluating the Bug Oracle 39
5.1 Detecting vulnerabilities 40
5.2 Testing on example enclaves 45
5.3 Real world application 49
5.4 Conclusion 52

6 Conclusion 55
6.1 Research question 55
6.2 General conclusion 55
6.3 Answering the research question 56
6.4 Future work 57

A Fuzzer source code **59**

Bibliography **61**

Abstract

When executing a security sensitive application, a developer has to trust many systems, from hardware to software, to ensure the integrity and security of the application. The Trusted Computing Base (TCB) of these systems comprises of complex hardware and many millions lines of code, the security of which cannot be assessed. In order to solve this problem, Intel developed the Software Guard eXtensions (SGX). These extensions give the developer a possibility of running programs inside an isolated space within the processor memory, secure from even the highest software privilege levels, reducing the complexity to a hardware TCB. However, Intel SGX leaves memory management to the untrusted operating system and explicitly does not include side channels in the threat model. This opens the way for side channels as a means of extracting enclave secrets by monitoring enclave execution. Previous work shows that exploitation of vulnerabilities within enclaves is a well researched topic, but the detection and availability of vulnerabilities is needed to execute these attacks.

This work researches a way to automatically detect bugs in Intel SGX Enclaves by a means of fuzzing. Fuzzing is an automated means of finding vulnerabilities by randomising input until interesting behaviour is encountered. Fuzzing Intel SGX enclaves differs from the normal white-box fuzzing, but does not completely fall into the black-box fuzzing category. A bug oracle is developed by combining two available side channels: a page fault side channel and a page table entry bits side channel using access and dirty bits. Memory accesses inside an enclave are monitored with a 4KB granularity. By controlling the accessible pages to the enclave and keeping the working set small, detailed memory footprints are recorded and refined using the page table entry bits. These memory footprints can then be used to detect possible vulnerabilities. The types of detectable vulnerabilities and their influence on the memory footprint graphs are discussed. To enhance input generation by the fuzzer, the input space of enclaves is investigated and smart input for each input channel is discussed, resulting in better vulnerability detection rates. Comparing enclave runs with a chosen difference metric allows the user to find interesting enclave inputs and executions for further research.

Finally, an evaluation of the fuzzer and bug oracle is performed on enclave examples. The oracle is able to detect side channels with ease. Ultimately, the bug oracle is tested on a real world enclave, in which the tool was able to find a previously found bug in the edger8r tool from the Intel SGX SDK, showing the real world value of this tool.

Samenvatting

Bij het uitvoeren van een beveiligingsgevoelige applicatie, moet een ontwikkelaar meerdere systemen vertrouwen, van hardware tot software, om de integriteit en de veiligheid van de applicatie te waarborgen. De Trusted Computing Base (TCP) van deze systemen bestaat uit complexe hardware en vele miljoenen regels code, waarvan de veiligheid niet kan worden beoordeeld. Om dit probleem op te lossen, heeft Intel de Software Guard eXtenstions (SGX) ontwikkeld. Deze uitbreidingen geven een ontwikkelaar de kans om applicaties binnen een geïsoleerde ruimte binnen het processorgeheugen uit te voeren, beveiligd tegen zelfs de hoogste softwareprivilege niveaus, waardoor de complexiteit voor de TCB wordt beperkt tot de hardware. Intel SGX laat echter geheugenbeheer over aan het niet-vertrouwde besturingssysteem en neemt expliciet geen side channels op in het dreigingsmodel. Dit opent de weg voor side channels als een manier om enclave-geheimen te extraheren door de uitvoering van de enclave op te meten. Uit eerder onderzoek blijkt dat het aanvallen van kwetsbaarheden in enclaves een goed onderzocht onderwerp is, maar om tot dat punt te komen moeten kwetsbaarheden detecteerbaar en beschikbaar zijn.

Dit werk onderzoekt een manier om automatisch fouten op te sporen in Intel SGX enclaves door middel van fuzzing. Fuzzing is een techniek waarbij op een geautomatiseerde manier kwetsbaarheden gevonden worden door input te randomiseren tot het doelwit interessant gedrag vertoont. Het fuzzen van Intel SGX enclaves verschilt van het normale white-box fuzzing, maar valt niet volledig onder de categorie van het black-box fuzzing. Een bug-orakel wordt ontwikkeld door twee beschikbare side channels te combineren: een page fault side channel en een side channel dat gebruikt maakt van de toegang- en schrijf-bits in de page table entries. Het gebruik van het computergeheugen door een enclave wordt gemonitord met een 4KB-granulariteit. Door de toegankelijke pagina's te beheren en de werkset klein te houden, worden gedetailleerde geheugenvoetafdrukken vastgelegd en verfijnd met behulp van de invoerbits van de page table entries. Deze geheugenafdrukken kunnen vervolgens worden gebruikt om mogelijke kwetsbaarheden te detecteren. De categorieën van detecteerbare kwetsbaarheden en hun invloed op de geheugenvoetafdrukgrafieken worden besproken. Om het genereren van invoer door de fuzzer te verberen, worden de types invoer van enclaves onderzocht en wordt slimme invoer voor elk invoertype besproken, wat resulteert in betere detectiepercentages voor kwetsbaarheden. Door geheugenafdrukken met elkaar te vergelijken met behulp van een gekozen verschilstatistiek, kan de gebruiker interessante enclave-input en -output vinden en gebruiken voor verder onderzoek.

Ten slotte wordt een evaluatie van de fuzzer en het bug-orakel uitgevoerd op voorbeeld enclaves. Het orakel kan side channels gemakkelijk detecteren. Uiteindelijk is het bug-orakel getest op een real world enclave, waarin de tool in staat was om een eerder gevonden bug in de edger8r-tool van de Intel SGX SDK te detecteren, wat de meerwaarde voor deze tool benadrukt.

List of Figures

2.1 The storage of process pages in physical memory 6
2.2 Address translation schematic 7
2.3 A typical memory layout of an ELF executable 8
2.4 The Intel SGX enclave memory structure 11
2.5 The different ways of entering and leaving an enclave 12
2.6 An overview of the SGX-Shield tool chain 17
2.7 An overview of the attack presented in the Guard's Dilamma 19

3.1 A grey box model of an Intel SGX enclave 22
3.2 The different stages of a fuzzer 23

4.1 Functioning of the working set. 29
4.2 The page accesses of the ECALL function. 30
4.3 A typical memory layout of an enclave 31
4.4 Page faults of a running enclave 32
4.5 Enclave execution with a dynamic working window 33
4.6 Mapping the memory on the memory footprint 35
4.7 Cross checking the instruction pointer on the memory footprint. 36
4.8 Comparing multiple runs of the same enclave function. 37
4.9 Displaying access and dirty bit status on the graphs. 38

5.1 Situating the progress on a fuzzer template. 39
5.2 Different enclave memory footprint changes 42
5.3 Steps in detecting interesting enclave behaviour 44
5.4 The RSA `modpow` statemachine 47
5.5 Comparing enclave execution with changing secret. 48
5.6 Leaking enclave secrets using access and dirty bits. 50
5.7 Detecting the edger8r vulnerability 51

Chapter 1

Introduction

1.1 Motivation

When executing a security sensitive application, a developer has to trust many parties. A computer consists out of hardware components such as a hard drive, memory modules, the processor,.. On top of this hardware, an operating system provides an abstraction layer for applications so they do not need to directly interface with the devices. The operating system also provides the possibility of isolating and running multiple applications on one system. When writing a sensitive application, all these components have to be trusted to ensure the integrity and security of the application. The combination of these systems is made out of millions lines of code of which the security can not be assessed. When one vulnerability is present in these components that can be exploited by an attacker, the entire application becomes compromised. In order to solve this problem, Intel developed the Software Guard Extensions. These extensions give the developer a possibility of running programs inside an isolated space within the processor memory. This way, even when the host system is compromised, an attacker would be unable to extract secrets from the security sensitive application. Although Intel SGX has not seen a lot of adaptation by the industry, a lot of research has been done on the security properties of Intel SGX, as will be further discussed. This thesis tries to improve on that research by creating a fuzzer for Intel SGX enclaves.

1.2 Goals of this thesis

Although the main goal of Intel SGX is providing isolation and security for applications running inside the enclave, it turns out that there are some methods available to reveal information about what is happening inside an Intel SGX enclave. Side channels, which are a means of collecting information from an application using unintended information leaks due to the implementation of the system, are not included the threat model of Intel SGX. Side channels can be for example timing based, power consumption based or even sound based. This, together with the observation that insecure code running inside an Intel SGX enclave remains insecure,

leads to developers needing to be careful about enclave development.

A basic example of a side channel exploitation can be imagined in a faulty pin terminal in a bank where user is asked to input a secret pin number. The terminal is placed behind a wall and a potential attacker is unable to see what is happening inside. A side channel in this situation could be the terminal having different sounds for each keypad number. An attacker, unable to see which keys are pressed by the victim, can use the sound of the key presses to deduce the secret pin.

Finding vulnerabilities by manual code review is hard and requires a lot of work. To help the detection of these vulnerabilities, a technique known as fuzzing was developed. Fuzzing automates the discovery of vulnerabilities by repeatedly testing an application with changing input data and monitoring the results. With fuzzing, bugs and undefined behaviour in applications can be automatically detected, which then can be further investigated by the developer. Fuzzing has helped finding many previously unknown bugs in many software frameworks.

Intel SGX provides, at first sight, a black box for services to run in. Because of this and the different nature of Intel SGX enclaves, fuzzing tools developed for system software are not immediately applicable to enclaves. If no monitoring is possible, fuzzing Intel SGX enclaves is a difficult problem. Like the faulty pin terminal in the bank, Intel SGX prevents anyone from looking behind the wall. The goal of this work is to find a way to monitor the execution inside enclaves while the security guarantees of Intel SGX stay active. Can we find a method to shine light on the potentially black box enclaves and make them more grey box?

The goal of this work is to make a fuzzer that will be able to work with Intel SGX Enclaves. The fuzzer will generate changing input for the enclave, hoping to trigger a reaction inside the enclave. Using side channels, which are not included in the threat model of Intel SGX, the enclave can be monitored in a limited way to detect these reactions and possible detect vulnerabilities. If such a method can be found, what kind of vulnerabilities can be found using the fuzzing tool? Finding the differences and limitations of fuzzing enclaves is a key question in researching this topic.

1.3 Overview

Chapter 2 starts with necessary background information about operating system concepts and the relevant inner workings of Intel SGX. The chapter also contains previous work on Intel SGX, starting with a few papers detailing attack and defence against Intel SGX Enclaves. These sections serve as a way to get more familiar with the concept and working of Intel SGX. Finally, a few relevant papers about previous work on Intel SGX side channels are presented.

The next chapter, Chapter 3, introduces the concept of fuzzing, and why further research is can be useful in this topic in the context of Intel SGX Enclaves. The problem statement is also laid out at this point.

Chapter 4 explains the challenges around and the development of a bug oracle for Intel SGX enclaves. The bug oracle is evaluated and used in the following chapter,

Chapter 5, on the ability of detecting vulnerabilities. Chapter 5 starts with possible methods to detect vulnerabilities using the bug oracle, and then continues with an evaluation of the bug oracle is discussed, then an evaluation of that method on example enclaves is performed, as well as to detect a real world vulnerability in the Intel SGX SDK toolkit.

Chapter 2

Background and related work

2.1 Operating System concepts

2.1.1 Computer architecture

A modern computer consists of many devices that together form the hardware of a computer. The two main components of this hardware are the CPU and the memory. In this thesis, we will talk about the Intel CPU, more specifically the desktop and server versions of the CPU. Intel SGX is a technology implemented on recent Intel CPUs starting from 2015. The Intel CPU does not only processes data, but has a myriad of other responsibilities like caching, handling software privilege levels, interrupting and address translation. These concepts will be explained further down. The Intel CPU provides multiple logical processors to the computer. These logical processors can be used for multithreading. Memory is another crucial element to a modern computer. Dynamic Random-Access Memory (DRAM) serves as a fast accessible location to store program data and code. DRAM is different from conventional storage in that it is an order of magnitude faster, more expensive and volatile. The memory and CPU are managed together by system software.

2.1.2 Program memory

A process running on the operating system has access to a virtual memory space. Working with a virtual memory space, instead of a direct access to the physical memory makes everything considerably less complex. Every process running on a computer has its own virtual memory space: this way it is isolated from other processes. This isolation ensures that a rogue process does not get access to the memory space of another enclave. Furthermore can the virtual memory space be used to map IO devices and memory addresses of other processes running.

To achieve this virtual memory space, there is a need to translate the virtual addresses to physical addresses in the memory, using *address translation.* This translation is taken care of by the Intel processor using a special register and hardware. This removes complexity from the process running and enforces the virtual memory space isolation.

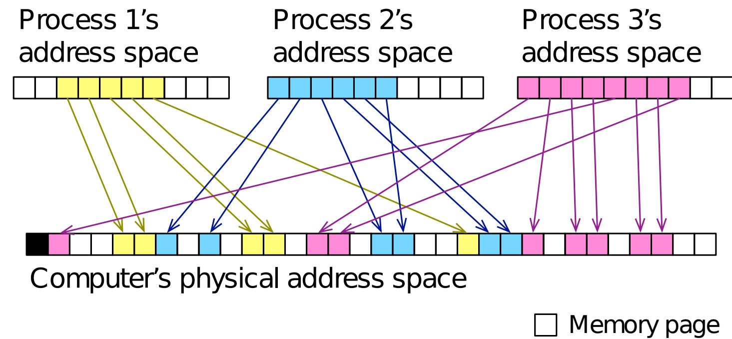


Figure 2.1: Pages from multiple processes address spaces can be efficiently stored in the computer's physical address space. (source [9, figure 11])

2.1.3 Memory pages

To translate the virtual memory address to the physical memory address, a way to translate the addresses is needed. The address translation is called *IA-32e* in the Intel Documentation. The system uses paging to achieve efficient translation. [14] The virtual memory of a program is divided into sections with a size of 4096 bytes. (The page size can be different for other processors) Only the pages that are necessary are loaded into memory by the operating system when they are needed by their process. Using pages, an operating system can efficiently make use of its physical address space by keeping only a necessary set of pages for each process in memory, as seen in figure 2.1. Addressing the pages is done with address translation. An index of pages is kept in the Page Directory. When a virtual page needs to be accessed, its *Virtual Page Number* is translated by a lookup into its corresponding *Page Frame Number* (PFN). Pages are kept in the Page Directory in leaves called *Page Table Entries* (PTE). A PTE contains the Page Frame Number and a few management bits containing information about permissions and for memory management usage. The lower part of the memory address is called the *Page Offset* and is left alone in the address translation. This offset stays the same for the virtual or memory page. A visual representation of address translation can be seen in figure 2.2.

Page table entry bits The page table entry bits provide the operating system with administrative information about the pages. The first important bit is the Present (P) bit. This bit indicates the presence of a page in virtual memory. Should a page not be present, a page fault is thrown and the page is retrieved from storage. If there is no more space for this page, another page needs to be discarded. This process is called *Page swapping* or paging.

Two other bits present in the Page Table Entry are the *access* (A) and *dirty* (D) bits. The access bit (or reference bit) is used to track accesses to a page, and the dirty bit is set when the data on the page has been modified since the page was loaded into memory. A memory management program can use these two bits to make an efficient paging algorithm. The access and dirty bits are not cleared by the

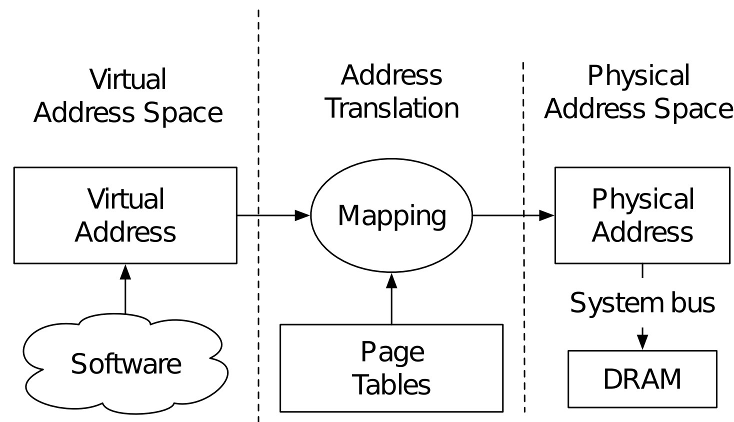


Figure 2.2: This schematic shows the role of address translation in accessing the physical memory. A virtual address is translated into a physical address by looking up the mapping in the Page Table Directory. ([9, figure 10])

processor. Instead, this task is left to the operating system. [14]

Finally, there are bits that indicate the permissions of a page. A *Writable* and an *Executable* bit let the processor know whether the page can be written to or whether the page can be executed from. These bits can be used to protect pages and ensure a faulty program can not accidentally do unintended things. Finally, there is the *Supervisor* bit, which disallows processes running in Ring 3 from accessing these pages.

2.1.4 Software privilege levels

To prevent a program from breaking the virtual address memory space, software privilege levels are introduced. [14] Intel processors use Ring 0 and Ring 3 as respectively highest and lowest privileged execution modes. Higher level privilege modes exist when taking the Virtual Machine Extensions (VMX) into account, where there is a higher privilege level for the host system (hypervisor) called *VMX Root* and a privilege level for the guest machines called *VMX Non-Root*. The system kernel generally runs in Ring 0. Certain data structures in memory like the page table control the process boundaries and interrupt tables. Only processes running in Ring 0 have access to these regions or instructions. Evidently, a program running in Ring 3 should not be able to switch to a higher privilege mode. To achieve certain functionality, a process can make use of system calls.

2.1.5 Faults

Faults are unintended events happening on which a higher privileged process is required to solve. Several types of faults exist, but the most important ones are:

General Protection Fault A general protection fault (#GP) happens when a certain action takes place where the process does not have the privilege for.

Page Fault When a page is not loaded into memory, a page fault (#PF) occurs. During a page fault, control is given back to the operating system and the operating system tries to load the page into the memory. It is possible to catch page faults by defining a page fault handler.

2.1.6 Executable layout

The layout of an executable binary is separated in multiple parts called sections. These sections divide a program in different parts containing each different kinds of data. The most important ones are described in the following list, taken from the ELF format specification [7].

`.bss` This section is intended to hold uninitialized data in the virtual memory space and is initialized with zeros on program initialization.

`.data` This section holds data used by the program.

`.rodata` This section holds read only-data and is meant to be placed in a non writable part of memory.

`.text` This section holds the executable instructions from the program.

Distinct from these sections present in the executable file, the heap and stack are both structures present in the executable layout during execution. A typical memory layout can be seen in figure 2.3.

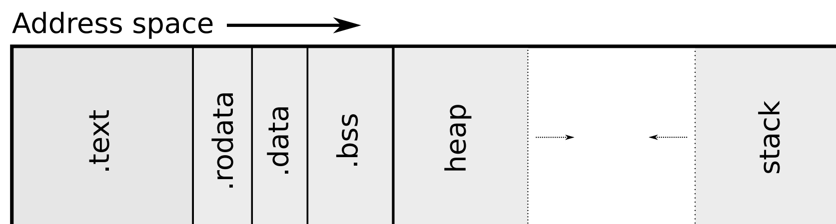


Figure 2.3: The memory layout of a typical ELF executable.

2.2 Software security

When writing an application, it is important to write code that is safe. Today's vulnerabilities can be put into three broad categories: memory safety vulnerabilities, side channel vulnerabilities and speculative execution vulnerabilities.

2.2.1 Memory safety vulnerabilities

Memory safety vulnerabilities can occur by writing unsafe code in unsafe languages. Unsafe languages are languages which give the developer low level access to the

memory and do not check whether the memory accesses are correct. These vulnerabilities can be divided in multiple categories. Spatial memory vulnerabilities are vulnerabilities which override bounds and access memory beyond the intended structure. These vulnerabilities happen for example when an index goes out of bound or wrong pointer arithmetic is used. Temporal safety errors occur when there is memory access to a structure that should no longer be available to the program. Use-after-free and double frees are examples categories of temporal errors. [21] A final type of memory safety vulnerabilities is an insecure usage of API functions from for example the standard c library, like the well known format string vulnerabilities. [22]

Exploitation of these memory safety vulnerabilities is a well researched topic. Many papers have and are being written about exploitation of and defence against these type of vulnerabilities. [11, 12, 34]

2.2.2 Side channel vulnerabilities

Another class of vulnerabilities are side channel vulnerabilities. Unlike memory safety vulnerabilities, side channel vulnerabilities are not immediately caused by a direct mistake by the developer. Side channel vulnerabilities are generally defined as a way to get information about an application by measuring the implementation of an application on the system it is running on. [37] A side channel vulnerability is not directly visible in the application code, but emergent from the behaviour of the application. There are multiple types of side channels (e.g. time based, power consumption based, sound based,..). Due to the nature of side channels, it can be possible to use side channels to reveal information over process boundaries. [4].

In this thesis, a memory side channel will be used. This side channel is due to how Intel SGX manages memory, and not a direct fault of the enclave developer. A developer can however take some side channels into account when developing an application and implement defences. [8, 27].

2.2.3 Speculative execution

Recently, a new class of vulnerabilities has been discovered. Speculative execution vulnerabilities exist due to optimizations within processors, where side effects of out-of-order execution can be used to read memory of other processes and break through the memory isolation assumptions security systems rely upon. The Meltdown and Spectre attacks, and more recently the Foreshadow paper, which is directly relevant to Intel SGX, demonstrate the significance of this category of vulnerabilities. [5]

2.3 Intel SGX concepts

Intel Software Guard Extensions (SGX) is a technology implemented on the Intel processor desktop and server line since 2015. Intel SGX provides enclaves in which software can be run, safe from any software running on the computer. This guarantee is enforced by the Intel processor itself. Similar solutions also exist for other architectures, such as TrustZone for ARM [1] and Sanctum for RISC-V architectures [10].

The information in this section was retrieved from the paper "Intel SGX Explained" [9] and others [3].

2.3.1 What is Intel SGX

Intel SGX enables users to protect a part of memory space, on which a program can run isolated from the other processes. The Intel SGX Enclave system is designed to be easily usable for application developers. Code should be able to be easily ported to Intel SGX without major modifications by a developer. One of the methods to archive this is letting the enclaves behave like external libraries to running processes. A running application can access the enclave interface with a wrapper generated by the Intel SGX SDK. The enclave is placed in a memory range in the virtual address space of the host application (the application that loaded the enclave), but the host application is prohibited by the Intel processor to access the enclave memory. The enclave memory on the other hand is able to read and modify memory outside its boundaries.

Enclave memory management

The reason the host application is unable to access the enclave memory is due to the region reserved by the Intel Processor called the *Processor Reserved Memory* (PRM). The Processor Reserved Memory is only accessible by the Intel processor, who enforces this during address translation. Intel SGX Enclaves are loaded in this part of the memory, in a region called the *Enclave Page Cache* (EPC). Due to the constraints by the Intel processor on the PRM, nothing can read or write to the EPC.

The EPC contains the pages used by the different enclaves, similar to the page system used by the operating system. Remarkably, Intel SGX requires the operating system to manage the Enclave Page Cache. For security purposes, extra metadata about the enclave pages is kept in the *Enclave Page Cache Map* (EPCM). Should the operating system try violate the security of an enclave, the EPCM can detect this intrusion and prevent it. One of the important pieces of information kept in the EPCM are three permission flags. Enclave pages can be marked ad readable, writable and/or executable. These values are indented to be set by the enclave developer and overwrite the operating systems defaults. A visual representation of the enclave memory structure can be seen in figure 2.4.

Because the PRM region cannot be written to by the operating system, the operating system cannot load the enclave pages directly to the EPC. To load an enclave, special steps have to be taken.

Enclave memory structures

When an enclave is initially created, just before being loaded, a *SGX Enclave Control Structure* (SECS) is created. The SECS contains metadata about the enclave and is stored away from the enclave memory, which makes it also inaccessible by the enclave. One of the attributes the SECS contains is the enclave debug flag. The

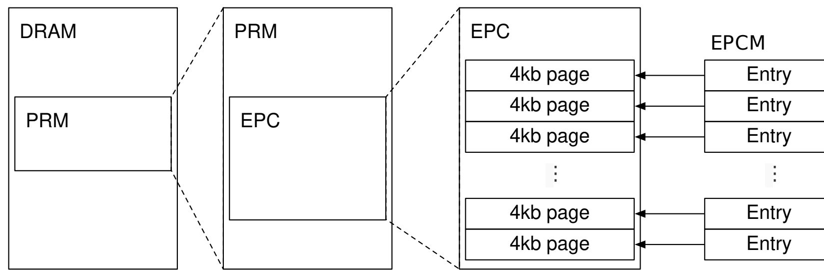


Figure 2.4: The Intel SGX enclave page cache is located within the processor reserved memory. The enclave page cache map keeps track of which pages are loaded. (source [9, figure 60])

enclave debug flag enables the debugging features of SGX, essentially meaning that the enclave contents can be read using a special procedure. This mode is meant for development and is set by default. To unset the debug flag, a special signing key from Intel is needed. This means that Intel controls which enclaves are allowed to run in production mode.

SGX enclaves support multithreading using the *Thread Control Structure* (TCS). The TCS enables multiple threads to run concurrently within an enclave. The TCS is stored on a page in the EPC, but due to restrictions on the page, the enclave is unable to directly access the page that stores the TCS.

Another frequently used structure is the *State Save Area* (SSA). Whenever the enclave encounters a hardware exceptions like interrupts or page faults, the execution context is saved to the SSA before handing over control to the exception handler in the operating system. The execution contains information needed to resume execution after the fault is handled. The SSA is like the TCS also stored in the EPC. When giving control back tho the operating system, some registers are scrubbed to prevent potentially leaking information. For example, on a page fault, the location within the requested page is removed by clearing the lower bits.

Enclave life cycle

An SGX enclave is created using the `ECREATE` instruction. This instruction initializes the SGX Enclave Control Structure and sets the size and base address of the enclave. After the initialization, the enclave pages are loaded into the EPC using the `EADD` instruction. `EADD` accepts a special page information (PAGEINFO) structure, from which it gets the information about which data has to be loaded. This way, pages are loaded into the EPC, which is normally inaccessible by anything but the Intel processor. After loading every page to the EPC, a special enclave called the *Launch Enclave* is used to obtain a `EINIT` token. This token is needed to launch the enclave.

Unloading enclaves is done using the `EREMOVE` instruction. This instruction frees the EPC by setting the enclave pages in the EPCM to available. After this is completed, the SECS page is freed and the enclave is completely unloaded.

Entering and leaving an enclave

Entering and leaving the enclave is abstracted from the developer using ECALLs and OCALLs. Every ECALL and OCALL needs to be defined in a separate file in the Enclave Definition Language (EDL) syntax. This file is parsed by the edger8r tool from the Intel SDK, which is used to generate the routines around the enclave application. An overview can be seen in figure 2.5.

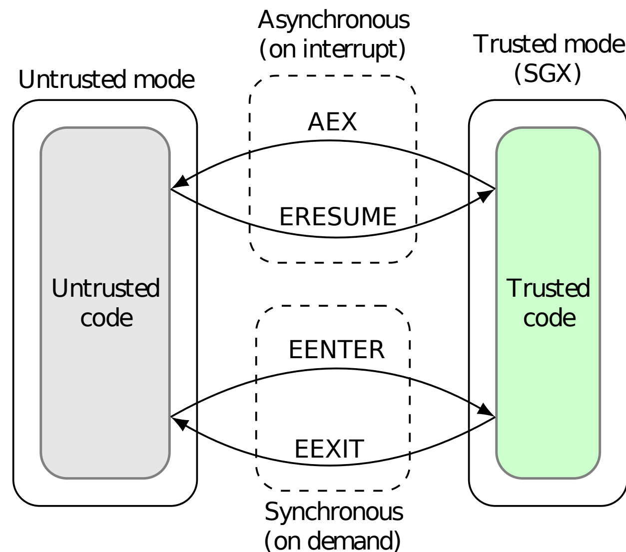


Figure 2.5: The different ways of entering and leaving an Intel SGX enclave. (source [3, figure 1])

The edger8r tool expects ECALLs and OCALLs to be annotated with arguments, denoting the type of the argument that is passed to the enclave. This information is then used to generate marshalling functions that load data into the enclave before continuing execution. The EDL syntax allows the declaration of simple data types like `int`, `short` and `float`, but also structures, enums, unions and pointers. To handle pointers, an additional pointer attribute has to be declared: `in`, `out` or `user_check`, depending on the pointer being the source or destination for the data handled in the enclave. The `user_check` leaves the developer to validate the contents of the pointer themselves. [13]

ECALLs When calling a function of the enclave, the enclave is called using an `EENTER` instruction with the offset of the corresponding ECALL function inside the enclave. The `EENTER` instruction does not clear the registers, so any arguments can be passed into the enclave. The trusted runtime library inside the enclave then copies data structures inside the enclave before continuing.

OCALLs OCALLs allow code inside the enclave to execute untrusted functions in the host application. Enclaves are not permitted to make system calls, so the

OCALL mechanism can be used to let the host application handle system calls. After an OCALL, the execution within the enclave is resumed with a variant of the `ECALL` called `ORET`.

Exception handling Exception handling happens using a different method. The Intel SGX SDK leaves developers to catch exceptions within the enclave. When an exception occurs, the enclave exits with an asynchronous enclave exit (`AEX`). The state is saved in the SSA and control is turned over to the host system. There, a function in the untrusted runtime system calls the exception handler in the enclave using an ECALL. After the exception handling ECALL returns, an `ERESUME` is executed for the thread on which the exception occurred.

2.3.2 The threat model

The threat model used by Intel SGX assumes that the whole environment is untrusted. This includes the operating system, all application code and the hardware peripherals to the Intel CPU package. This means that devices like a storage device or DRAM are included in the untrusted part of the system. Components inside the Intel processor package, such as the CPU caches and cores, are deemed secure. The threat model of Intel SGX does not include protection against side channels or the software running inside the enclaves themselves. This last exclusion leads to a wide array of possible attack scenarios against enclaves. [9, 25]

2.4 Previous work

2.4.1 Dark-ROP

In the paper "Hacking in Darkness: Return-oriented Programming against Secure Enclaves" [16], the authors explore a method of enclave exploitation they call Dark-ROP.

Return-oriented Programming attacks

A *Return-oriented Programming* (ROP) attack is an attack which hijacks the execution flow of a program by taking control of the stack. By carefully putting values on the stack, an attacker can piece existing application code together and execute unintended behaviour. These pieces of code are called *gadgets*. Normally, when a function is called, the return address is placed on the stack. When the called function completes, the return address is loaded back in to the instruction pointer and code execution resumes on the previous location. When an attacker gains access to the stack by for example a stack buffer overflow, the attacker can overwrite the return address with a return address of its own. Gadgets are small fragments of code with a return instruction at the end, which do certain behaviour like popping a value from the stack in a register. When the application code is known, an attacker can search

for the locations of useful gadgets and chain useful gadgets to further compromise the application.

Intel SGX and ROP

When dealing with Intel SGX, the normal procedure of finding gadgets inside the codebase becomes a challenge. The paper assumes the same threat model as this thesis: only the Intel processor is trusted and the enclave has the possibility of loading encrypted binaries, meaning that the enclave contents and thus also the locations of gadgets are unknown to the attacker. Dark-ROP answers to this challenge using the method described in this section. For the attack to work, the enclave must meet certain requirements.

Enclave assumptions For the attack to work, certain assumptions need to be fulfilled. The enclave needs to contain an `ENCLU` instruction. The `ENCLU` instruction is used, together with the value of the `rax` register, to determine the user space enclave leaf function to be called. The `ENCLU` instruction is needed to execute the `EEXIT` leaf function. The enclave also needs to contain pop gadgets for multiple registers. These gadgets needs to be available to load arguments for later function calls. Finally, a `memcpy` instruction is needed to copy memory in- and outside the enclave.

Finding the vulnerability First, a vulnerability leading to a stack buffer overflow must be present in the enclave. Using this vulnerability, the attacker can push data onto the stack until the enclave tries to access or return to an invalid page. This is detectable by checking the value returned in `CR2`, the control register that stores the Page Fault Linear Address (the address which caused a page fault) [3]. To prevent leaking the information from the enclave on an Asynchronous Enclave Exit, such as the position within the faulting page, Intel SGX clears the bottom 12 bits of the `CR2` register. Other general purpose registers are also cleared on an AEX. A possible value of `CR2` could be `0x41414000`. Finding the offset of this location in the stack lets the attacker redirect execution to an address of their choosing.

Locating pop gadgets After finding a way to redirect the control flow of the enclave, pop gadgets need to be found. Pop gadgets pop a value of the stack and stores it in a register. By setting up the stack in a particular way, the attacker can count the number of values popped from the stack by a gadget, before the return is called on an invalid address. By for example putting the values `0xF7741000`, `0xF7742000`, `0xF7743000`, `0xF7744000`,.. on the stack, the attacker can monitor the page fault oracle for the faulting address. When the enclave faults on for example `0xF7743000`, it can be deduced that there were 2 pop instructions in the current gadget. The page fault oracle does not reveal to which registers the values are popped.

Finding an enclave exit To execute the `EEXIT` leaf function, the `rax` register needs to contain the value `0x4`. The important part here is that the hardware does not clear the general purpose registers on exiting the enclave with `EEXIT`. Filling the stack with the value `0x4` and trying to encounter a `pop rax` gadget, the enclave is further searched until an `EEXIT` occurs. Once the location of the `EEXIT` instruction is found, the information revealed by the general purpose registers on enclave exit can be used to infer which registers the pop gadgets push their values to.

Compromising the enclave Finally, using these pop gadgets, the attacker searches for a `memcpy` function inside the enclave. By setting up the registers using the correct pop gadgets, it is possible to use the `memcpy` function to copy data to and from the enclave from attacker controlled memory regions. This access to the enclave then leads to a total compromise of the enclave security model.

Mitigation

The paper proposes some mitigation techniques for their Dark-ROP technique. One option is limiting the availability of useful gadgets by more carefully considering the context around `ret` or `ENCLU` instructions and trying to evade the `memcpy` function. Another possible mitigation is the implementation of fine grained Address Space Layout Randomization (ASLR). This solution is discussed in the SGX-Shield paper (see § 2.4.2).

2.4.2 SGX-Shield

The paper “SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs” [26] explores the possibility of adding *Address Space Layout Randomization* (ASLR) to Intel SGX enclaves. ASLR is a frequently used technique where applications, libraries, the stack and the heap and are loaded on a random address within the virtual memory space. This makes it difficult for an attacker to locate the destination of possible jumps while exploiting a vulnerability. Normally, because the size of the virtual address space, ASLR randomizes the loading offsets for memory elements. Because the available space in an Intel SGX enclave is very limited in comparison, a finer type of ASLR is needed.

Challenges

The authors define 4 challenges that have to be solved in order to get SGX-Shield operational. Firstly, Intel SGX exposes the memory management and layout to untrusted system software. This means that SGX-Shield needs to protect the application running inside the enclave from leaking information through these side channels. Keeping the limited memory space in mind, the degree of randomness for the randomization is limited. Another challenge is the relative addresses within de application code. Because of the enclave attestation, updating the relative addresses before loading the code into the enclave is not an option. Finally, some structures

within an Intel SGX enclave cannot be moved and have fixed addresses. The paper assumes the same threat model used in this thesis: only the Intel Processor is trusted and all other hardware and software components are not.

Solutions

In order to address these challenges, SGX-Shield uses the following elements to achieve a fine grained ASLR within an Intel SGX enclave:

Multistage loader Because the operating system is completely untrusted, every ASLR operation has to happen within the enclave. For this, SGX-Shield uses a multistage loader, consisting of 3 steps. First, the application is compiled using the SGX-Shield tool chain. This prepares the additional steps of the multistage loader and prepares the application for the fine grained ASLR. Second, the loader loads the application into an Intel SGX enclave and finally it starts loading the application into the memory space from its pages, using a securely randomly generated number from within the enclave. After loading the application, the multistage loader removes itself from the application and jumps to the starting point.

Fine-grained randomization In order to achieve fine-grained randomization, the application code from the target program has to be divided in smaller parts. SGX-Shield uses 32 or 64 byte *randomization units*. Every unit is modified to jump to the next randomization unit instead of continuing execution in the spatially next randomization unit. The units are randomly loaded into the code pages of the enclave, and every absolute address is modified to point to the right randomization unit. The stack is placed on a random base address within the data pages of the enclave, and the heap and global and static variables are placed randomly over the remaining data pages.

Software DEP Software *Data Execution Prevention* (DEP) is a method of ensuring that no data pages are executed as code, and no code pages are modified as data. Using this protection, vulnerabilities which load executable code into memory structures are unable to execute them. There are two kinds of memory access the paper addresses. Explicit memory accesses and stack accessing instructions. Explicit memory accesses are handled by a *Non Readable and Writable boundary*, or NRW boundary in short. By keeping all the modifiable data above a certain address (the NRW boundary), a technique can be used to enforce all memory accesses operands to point above the NRW boundary. This is done by keeping the address of the boundary in the `r15` register. Every memory address is surrounded by a check, which enforces the boundary. For the second type of memory access, the stack accessing instructions, SGX-Shield ensures that the stack pointer can never point to the code pages. To handle implicit changes in the stack pointer, guard pages are used. These pages are placed around the stack and marked as inaccessible. Accesses to these guard pages generates a fault.

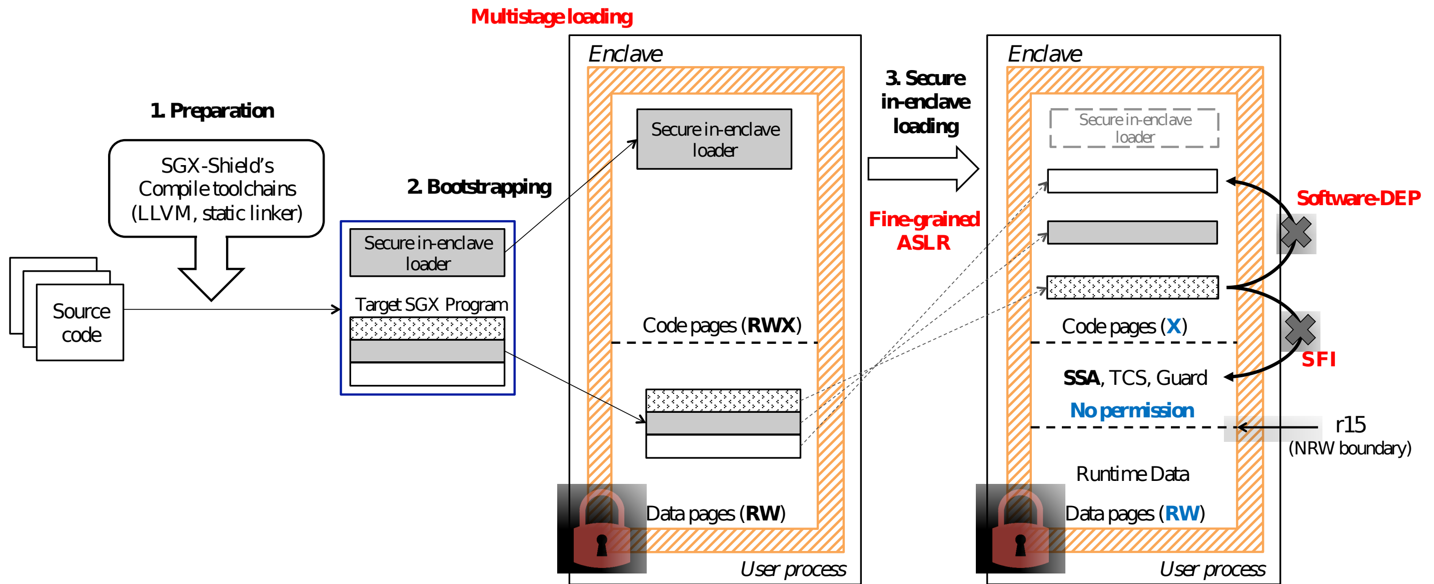


Figure 2.6: An overview of the SGX-Shield tool chain discussed in this section. (source [26, figure 2])

A final defence used by SGX-Shield is aligned control transfer. SGX-Shield enforces that execution starts at the top of randomization units. Access to to the security critical data placed on known positions within the enclave is prohibited by placing the State Save Area, which is not relocatable, below the NRW boundary. This prevents reads and writes to the SSA.

An overview of the SGX tool chain can be found in figure 2.6.

2.4.3 Guard's Dilemma

The paper "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX" [3] explores an Intel SGX specific attack taking control of an enclave after the exploitation of a vulnerability in an enclave. The attack described uses a stack overflow vulnerability or the ability to corrupt function pointers. After an initial control flow hijack, the attack resembles a Return-Oriented Programming attack, by combining 2 powerful gadgets present within the Intel SGX SDK trusted library. The attack has no need for pop gadgets, which for example Dark-ROP does (§ 2.4.1). This makes the amount of needed code for successful exploitation limited and simplifies the attack.

Assumptions

The paper weakens the usual threat model where an attacker has complete control over the machine, and only needs the attacker to have compromised the application that hosts the enclave. The attacker needs a memory corruption vulnerability, the ability to create fake structures, the location of the SGX enclave inside the virtual

memory space and knowledge about the enclaves binary. The paper furthermore assumes the Intel SGX SDK was used in creating the enclave, and that the memory is additionally hardened by applications like SGX-Shield. (see § 2.4.2)

Attack components

In order to mount the attack, 2 primitives from the Trusted Runtime System (tRTS) are used: the ORET and the CONT primitive. Using these primitives, the attacker is able to set the CPU registers to chosen values.

ORET primitive The ORET primitive uses a function called `asm_oret`. This function is used to restore CPU context after an OCALL. In order to use this primitive, the attacker needs control over the instruction pointer to redirect control flow to the `asm_oret` function and the ability of modifying the stack contents. The ORET primitive gives control over some of the CPU registers, including the instruction pointer and the `rdi` register, which holds the first function argument.

CONT primitive The CONT primitive uses the `continue_execution` function from the tRTS. This function is used to restore the CPU context after an exception. To use the CONT primitive, the function `continue_execution` function needs to be called with a user controlled `rdi` register. The CONT primitive allows the attacker to set the contents to all general purpose CPU registers.

ORET+CONT loop Combining the ORET and the CONT primitives, a ORET+CONT loop can be created. The CONT primitive is repeatedly used to invoke gadgets with chosen register values and manipulating the stack pointer to attacker controlled memory, while the ORET primitive is then used to set the `rdi` register and invoke the CONT primitive again.

Attack execution

The preparation of the attack consists of finding gadgets in the code that can be used. As it is difficult to randomize all code, solutions like SGX-Shield leave some code non-randomized. Next, a gadget chain is constructed together with the needed register states. Next, multiple fake structures have to be created that contain the registers for the CONT primitives, as well as a fake stack that supports the ORET+CONT loop. Because enclaves can access memory outside enclaves, this structures can be located in the attacker controlled memory space.

Using the ORET+CONT loop, the attacker can write code to the enclave and take complete control over the enclave. An overview of the attack can be found in figure 2.7.

2.4.4 Stealthy Page Table-Based Attacks on Enclaved Execution

The paper “Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution” [31] discusses a novel side channel attack on Enclaves.

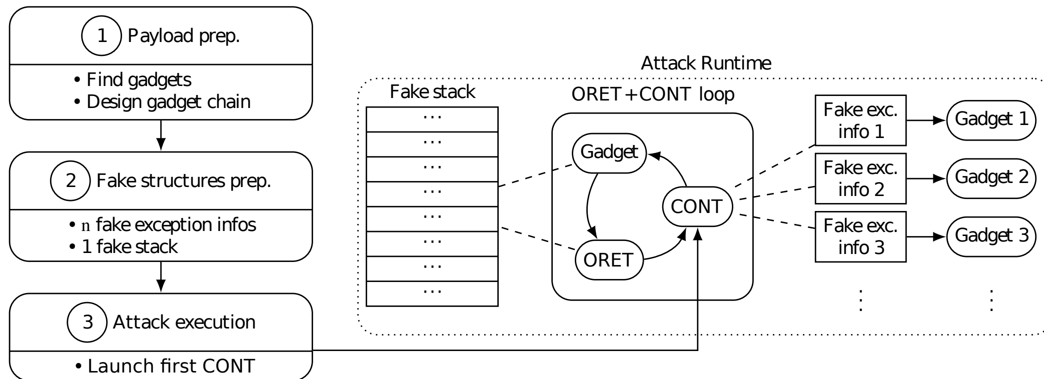


Figure 2.7: An overview of the proposed attack presented in the Guard's Dilemma. (source [3, figure 4])

Instead of limiting the access rights to enclave pages, the side channel is able to infer page accesses without using page faults. The technique used for this involves monitoring the dirty and access bits present in the enclave page table. These bits are still updated on enclave execution. For the monitoring to happen frequently, the enclave can be interrupted using Inter-Processor Interrupts, a system in the Intel processor used for synchronization across cores. A second method for measuring page accesses is also proposed, by measuring the delay of accessing the PTE while using a technique called `FLUSH+FLUSH`. The resulting page access patterns then can be used for further investigation. The authors show that the technique is able to extract session keys from a real world cryptographic library function.

2.4.5 SGX-Step

In the paper "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control", the authors present a framework to monitor enclave execution at an instruction resolution. This is done by using the Advanced Programmable Interrupt Controller (APIC) timer to schedule interrupts after each instruction run by the Intel SGX enclave. By carefully choosing the duration of the APIC timer and attaching a kernel module to the APIC event callback, SGX-Step provides a tool to step through an enclave on an instruction basis and execute or measure along the way. This then can be used for further more precise exploitation of side channels attacks on Intel SGX enclaves.

2.4.6 Preventing Page Faults from Telling Your Secrets

The paper "Preventing Page Faults from Telling Your Secrets" [27] presents the problem that many applications are vulnerable to page fault side channel attacks. They propose a solution which makes the execution flow of programs deterministic. Regardless of the input or the secrets, an application should follow the same page fault sequence. They achieve this in software with a technique the authors call

deterministic multiplexing, where sensitive code branches are kept on the same page. Their solution also provides solution for code branches that are larger than a single page. Finally, a possible route for hardware defence is proposed, with the idea of contractual execution, where some pages are ensured to be always mapped to memory.

2.5 Conclusion

We have learned that by leaving memory management to the untrusted operating system, Intel SGX opened up a large possibility of using side channels. Although an attacker can not gain immediate access to the enclave memory contents, a lot can be learned from the inevitable information leaking through side channels. The Dark-ROP and Guard's Dilemma paper show the state of the art on exploiting vulnerabilities within an enclave. But for these attacks to be useful, a vulnerability needs to be present and found within the enclave. This is where a fuzzer is able to help. The next chapter discusses the basics of fuzzing and the applicability to Intel SGX enclaves.

Chapter 3

Fuzzing Intel SGX enclave execution

3.1 Problem statement

In order to find vulnerabilities in SGX Enclaves, we can make use of a fuzzing framework. There are however some problems that first have to be solved. At first sight, Intel SGX enclaves behave like a black box: the memory isolation assures that no memory can be accessed from outside the enclave and the ability of vendors to supply encrypted blobs to the enclaves means there is no information about the contents of the enclave. Apart from the parameter input to the enclave and the results/crashes returned by the enclave, there is little to monitor. A first intuition could be to approach Intel SGX enclaves from a black box perspective. Fuzzing Intel SGX enclaves this way is similar to fuzzing hardware or remote systems.

In practice however, there is more information available. Because Intel SGX does reside on the host system and leaves some functionality to the untrusted operating system, side channels can become a valid approach to extract valuable information about the enclave execution. Monitoring memory usage, CPU caches or even timing the duration an enclave spends time executing reveal some information to an attacker. This leads to the idea that Intel SGX can be approached as a grey box, which can greatly improve the results a fuzzer can achieve. A visual comparison can be found in figure 3.1.

3.1.1 Research question

In this work, we try to research an answer to the following hypothesis:

Finding vulnerabilities in Intel SGX Enclaves by fuzzing can be enhanced over black box fuzzing using side channel measurements.

The question now is which side channel measurements are usable in the development of a fuzzer. And if these measurements are available, what type of vulnerabilities can be found? How can we better supply input to an Intel SGX enclave? How is

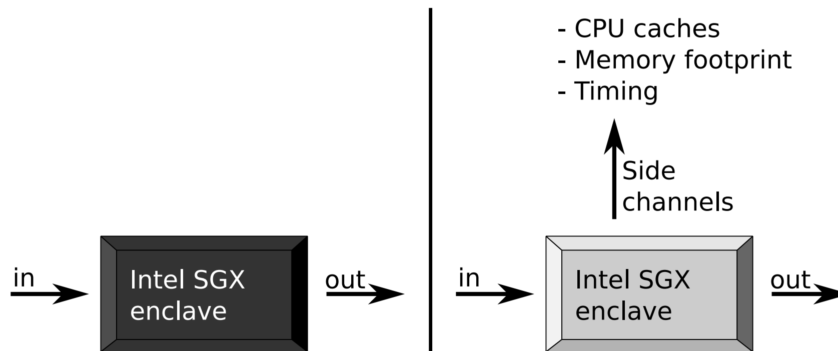


Figure 3.1: A black box model and a grey box model of an Intel SGX enclave. With a black box model, similar to hardware or remote systems, we have only access to the input and the output to the enclave. With a grey box model, side channels provide additional information about enclave execution.

fuzzing Intel SGX enclaves different from fuzzing normal systems? How automated can we make de discovery of these vulnerabilities by the fuzzer. The answers to these questions will be discussed in the next chapters. First, we take a detailed look at how fuzzers generally work.

3.2 Fuzzing concepts

Fuzzing is a technique to find vulnerabilities in software programs. The idea of fuzzing was introduced in 1990 [20] and has since seen an enormous development in the security community. The core concept of fuzzing is the automated search of bugs in software or hardware. To do this, multiple techniques were and are developed and still being improved on. One of the most well known fuzzers is American Fuzzy Lop [36], which has found numerous bugs in many libraries and applications. The paper "Fuzzing: Art, science, and engineering" [18] goes in great detail about the current state of fuzzers and research. This section is based on that work.

3.2.1 Terminology

A *fuzzer* is a program or library used to detect bugs in software systems. The fuzzer itself launches the target application and provides input to that application. Depending on the behaviour of the application, the fuzzer can detect interesting behaviour and modify the input to the application.

Practically every program consists of multiple functions and branches in the codebase. The idea of *coverage* in the fuzzing space refers to the amount of application that was reached by the fuzzing campaign. Keeping track of the application coverage ensures that the whole application is tested and not only some parts. Coverage can be difficult or impossible to monitor. As coverage is important, multiple techniques have been developed to interpret execution information as coverage information.

Finally, a *fuzz configuration* denotes the set of rules a fuzzer uses for a certain fuzzing campaign. The fuzzer configuration determines the used strategy and input generation.

3.2.2 Difference between black/white box fuzzing

In general, the targets of a fuzzer can be divided in 3 large sections. The most powerful form of fuzzing is white box fuzzing. In white box fuzzing, the fuzzer has full access to the target system or application. This means the fuzzer has access to the source code or binary and the system running the application. White box fuzzers employ every piece of information they can get and can for example add instrumentation during compilation time of the target. Fuzzing these kinds of targets is transparent, thus called *white box*. Black box fuzzing is the complete opposite of white box fuzzing. In a black box system, the fuzzer has no access or extremely limited information about the target system. The fuzzer tries to guess input to the target application until unexpected behaviour manifests. In the middle of these two extremes is grey box fuzzing. In grey box fuzzing, the fuzzer does not have complete access to the target system, but it can still receive some and imperfect information about what is happening in the target application. Intel SGX initially presents itself as a black box system, but this work will show that information channels are available and can lead to a better approach of Intel SGX enclave fuzzing.

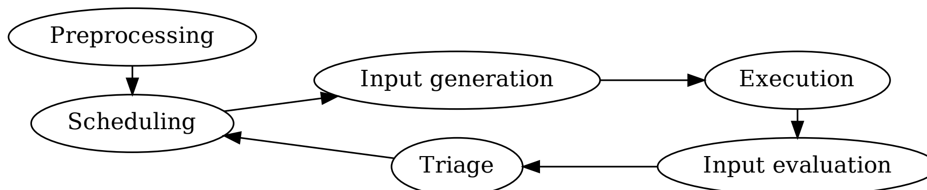


Figure 3.2: Most fuzzers follow the same pattern of steps while working on a fuzzing campaign. After preprocessing, fuzzers enter a continuous loop of mutating the input to create better test cases and executing the target with the generated input.

3.2.3 How do fuzzers work

The fuzzing process contains in general several steps which return in most fuzzers. These are: preprocessing, scheduling, input generation, input evaluation and configuration updating. [18] After preprocessing, fuzzers enter a continuous loop of mutating the input to create better test cases and executing the target with the generated input. A graphical representation can be seen in figure 3.2. The fuzzer in this thesis will not implement the preprocessing and scheduling steps, but will discuss the input

generation and evaluation in Chapter 5 and the bug oracle used to evaluate the input in Chapter 4.

Preprocessing Before starting the fuzzing of a target, there is the option of preparing the fuzzing campaign. A common technique used by many fuzzers like the popular AFL fuzzer is instrumentation. The idea of instrumentation is adding small pieces of code to the binary around function calls and other interesting places, as to report back to the fuzzer about the state of the execution. Both static instrumentation, where the instrumentation is added during the compilation phase, as dynamic instrumentation, where the instrumentation is added during execution exist. The information retrieved by the instrumentation can than be used for more efficiently modifying the input.

Another preprocessing technique is preparing a model for the target application input. Some fuzzing campaigns can result in better coverage and results when the fuzzer has a good model of the input that the target expects.

Scheduling Scheduling is the idea of choosing a fuzzing configuration which results in the best performance of finding bugs. Some fuzzers only keep one configuration, which makes scheduling straight forward. Other fuzzers like AFL use evolutionary algorithms to choose a configuration.

Input generation Input generation is one of the main functions of the fuzzer. As the input to the application is the last step before the execution, the result of an execution and finding eventual bugs is completely dependent on the input generation. There are mainly two kinds of input generation used in the majority of the fuzzers: model based input generation and mutation based input generation. Model based fuzzers use a model to generate possible inputs. An example would be a packet fuzzer which fills the fields of a packet with data, but calculates a correct checksum for the packet. The model is usually prepared in the pre processing step. Mutation based fuzzers explore the input space by mutating previous inputs. These mutations can be based on bit-flipping, arithmetic mutation, block-based mutation and dictionary based mutation. [18]

White-box fuzzers can use completely different techniques for input generation, as they have access to the source code or binaries of the target application. Symbolic execution is a technique where every possible path is taken and investigated by using a more arithmetic approach and calculating the possible values for arguments. Apart from symbolic execution, guided fuzzing is also a commonly used technique where the fuzzing is guided by a previous analysis of the code.

Input evaluation After execution of the target application with a carefully chosen input, the results of the execution have to be analysed. First and foremost, detecting an application crash is straightforward to detect. Some bugs however do not always lead to an application crash. To detect these, bug oracles are used. A bug oracle is a method which reveals and detects behaviour that may be the indication of a bug

being present. Bug oracles are used to detect memory and type safety, undefined behaviour and semantic differences in executions.

Triage During a fuzzing campaign, many input cases will be tested that lead to crashes or potential crashes. Triage is the selecting and minimizing of interesting test cases for further fuzzing. Some inputs may lead to the same crash, so reducing these test cases resulting in the same crashes is beneficiary to limit the amount of test cases for further fuzzing. Triage can also help to prioritize and minimize test cases by selecting the most interesting or smallest samples.

Chapter 4

Developing a Bug Oracle

4.1 Goal of this chapter

This chapter will discuss the main challenges of finding a bug oracle in Intel SGX enclaves. As discussed in the previous chapter, fuzzing can be divided in 3 main categories: black box fuzzing, grey box fuzzing and white box fuzzing. Intel SGX aims to be able to protect code running inside an enclave from any attacker with control over the operating system. Because of this, tools generally used to fuzz programs do not work on Intel SGX Enclaves. In this chapter, we will test the hypothesis if it is possible to find bugs inside enclaves with an automated tool using a bug oracle. To help us find these bugs inside a presumingly black box SGX program, a combination of available oracles will be used. Using these oracles, we can transform the black box nature of an Intel SGX enclave to a more accessible grey box one.

4.1.1 Desired properties of a bug oracle

To evaluate the bug oracle created in this chapter, 3 properties are important to keep in mind: determinism, granularity and efficiency. The *determinism* of a bug oracle leads to the quality and correctness of the gathered information. Page fault based side channels are more deterministic than timing based side channels like cache side channels, due to the more noisy nature of these systems. [17] The *granularity* of a bug oracle talks about the achievable precision in space and time components. Where page faults have a spatial granularity of 4KB, using cache lines achieves a granularity of 64 bytes. Granularity can also be about the temporal part of measuring, i.e. the time steps. Where frameworks like SGX-Step [29] achieve instruction based time steps, a page fault oracle can only measure once every page fault, which is dependent on the enclave code and data accesses. Finally, the *efficiency* of a bug oracle shows the usability of a bug oracle. A more efficient bug oracle is able to derive more information about a system using less resources. Finding a good balance between these 3 qualities requires making trade-offs.

4.1.2 Combining page faults and page table bits

Intel SGX enclaves are used with the promise of protecting the code running inside enclaves from the host system. To archive this protection, the enclave has to run in production mode. The developer of the application also has the possibility of loading new, encrypted blobs into the enclave, so no knowledge about the internal binary is available. Because of this, few methods exist to gain information from execution within these enclaves. There is no direct access to the enclave content so other means of gathering information are needed. One of this methods are side channels. Previous research [17, 27, 30, 35] has already explored the use of side channels in the context of Intel SGX enclaves. Side channels are a limited way of gathering information from a system, as they generally do not directly interface with a running application. In this chapter, two side channels will be discussed that when combined together allow for efficient detection of bugs in black box enclaves. This technique makes enclaves more transparent for continued research. First we discuss some concepts we will be using further in this chapter.

Working set

Any executing program is loaded into memory on a set of pages (§ 2.1.3). Intel SGX enclaves are also loaded into memory with added protection measures. But, SGX does require the operating system to handle the enclave pages. The set of pages that a program uses can be called the *working set* of that application.

For the purpose in this work, it will be beneficiary to make the working set as small as possible by making pages inaccessible. When the executing enclave encounters an inaccessible page, the operating system is asked to resolve this conflict. By having many inaccessible pages, also called *trigger pages*, we can collect useful information from the enclave execution. The size of the limited working set is called the *working window*.

Working window

A working window refers to the size of the working set, the set of pages that are accessible at a certain time. The accessibility of pages can be controlled by changing the read, write and execute permissions on a page basis. (see § 2.1.3) Here, the working set is implemented as a FIFO (first in, first out) buffer of a fixed size. Whenever a page faults, the faulting page is inserted in the first position of the working set buffer. The last entry of this buffer is then automatically removed. The size of the working window refers to the amount of entries in the FIFO buffer. A graphical representation can be seen in figure 4.1. Consider an initial working set with a working window of size 3 and with page entries p_1, p_2 and p_3. In this example, p_1 is the oldest entry in the buffer and p_3 the most recent. When a new page fault happens, the oldest from the 3 entries from the working set (p_3) is evicted. The new working set consists of pages p_2 to p_4. When a page is inserted into the working set, it should become accessible, meaning the program again has the ability to read,

write or execute from this page. When an entry leaves the working set, it will be made inaccessible, disallowing the executing process to access the page in any way.

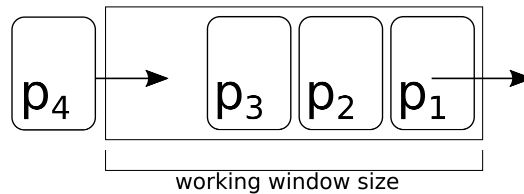


Figure 4.1: The working set behaves like a FIFO buffer with a fixed size. On a new page fault, the faulting page is inserted into the working set and the oldest page evicted. The working window size determines the amount of pages in the working set.

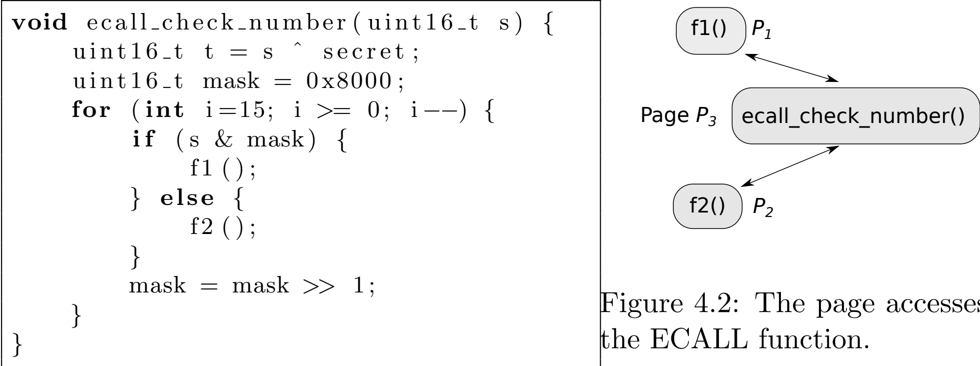
Memory footprint

Using the working window to limit the available pages to the executing program, every time a page not in the working window needs to be accessed, it faults. These pages can be called *trigger pages*, because they trigger a new page fault event. Keeping track of which pages fault, we can construct a memory footprint of the running program. This memory footprint gives us a coarse idea of which resources are being used. The amount of detail in this memory footprint is determined by a few factors. The two most important ones are the page size and the size of the working window.

With a page size of 4096 bytes [14], the space resolution of the memory footprint is limited to the size of the pages. To prevent page faults leaking the instruction pointer inside the enclave, Intel SGX clears the upper bits of the instruction pointer on a page fault. This way, the operating system can still know which page faulted, but not the location the running enclave tried to access within the page. Because of this, it is not possible to determine where in the page memory is accessed using page faults alone. But thanks to the deterministic property of page faults (a page always faults when it is needed), a memory footprint based on page faults does allow to determine certain behavior of the running program, even if it has only the space granularity of the page size (4K).

The size of the working window is also important to the detail of the memory footprint. With more pages in the working set, less trigger pages are hit. This leads to no page faults happening when execution or data access switches between different pages and losing potentially useful information. Keeping this window as small as possible is necessary for a detailed memory footprint.

Working window size example Imagine the following enclave function, inspired by an example in the Space 18 SGX tutorial repository [28] and [31, figure 3]:



```
void ecall_check_number(uint16_t s) {
    uint16_t t = s ^ secret;
    uint16_t mask = 0x8000;
    for (int i=15; i >= 0; i--) {
        if (s & mask) {
            f1();
        } else {
            f2();
        }
        mask = mask >> 1;
    }
}
```


Figure 4.2: The page accesses of the ECALL function.

This function receives an integer s, and then loops over the bits of the xor product of the argument `s` and an unknown secret `secret`. For every bit that is 1, the enclave executes `f1`. For every 0 the enclave executes function `f2`. If `f1` and `f2` are on different code pages and both inaccessible to the enclave, the enclave will page fault on every iteration when it tries to access one of the functions. However, if `f1` and `f2` are both in the working set, no page faults will occur and information about the enclave execution will be lost. This example also shows the strength of monitoring page faults. In the above function and looking at the memory footprint, the secret can be deduced with ease.

Access and dirty bits We can further improve the information of the memory footprint using the access and dirty bits from the Page Table Entries (PTE). Access and dirty bits are a tool for the operating system to do more efficient paging. (see § 2.1.3) These bits are updated on accessing or writing to a page. Monitoring the status of these bits makes it possible to detect if a page is accessed or written to during the enclave execution. In general, only pages that are in the working set are able to be accessed, so only the access and dirty bits from pages in the working set have to be monitored. Using the information these bits provide, the memory footprint can be improved.

When protecting pages, there is also the possibility of only revoking or allowing reading, writing or executing to a page. It is possible to build a complex algorithm to detect the different uses of pages by the enclave using only page faults, but using A/D bits is considerably less complex: on every page fault, the status of the pages in the working set can simply be read out from the page table entries. Using the A/D bits gives us information on the complete working set without requiring many more repeated page faults. This makes the PTE bits suited to supplement page faults.

Page offset terminology

In the following section, a lot of references within and to pages will be made. As this can get somewhat confusing, this paragraph gives a small overview of the existing terminology as used in the literature. As discussed in § 2.1.3, pages have a virtual and a physical address.

A *Page Frame Number (PFN)* is the index of a page in physical memory. This index counts the amount of pages, and not the total bit-offset of a page.

A *Page Number* is used to index a page in virtual memory. A Page Number can be used to look up the PFN of a page in the page table.

Page offset usually refers to the offset within a page. This offset is defined in bits.

For ease in discussing memory footprints for SGX Enclaves, the term *Enclave Page Number* will be used. The enclave page number denotes a page index starting from the start of the enclave. For example, in an enclave of size 2MB, there are 512 4kB pages. The fourth page will have enclave page number 3.

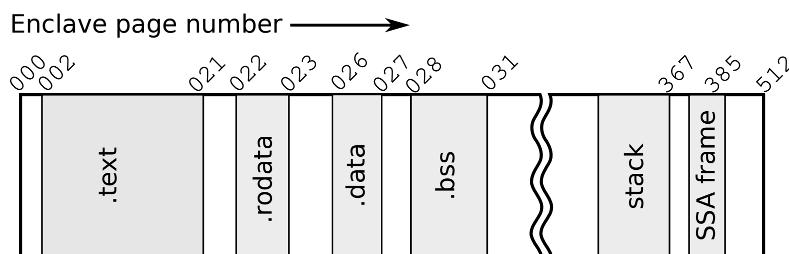


Figure 4.3: The memory layout of a typical enclave compiled by the Intel SGX SDK. The `.text`, `.data` and `.bss` sections are loaded into the lower part of the enclave memory, while the stack and the enclave SSA frame are present in the upper part of the enclave memory.

Enclave memory layout A typical memory layout of an enclave resembles that of any executing program. Using `objdump` on an enclave file we can extract the different memory sections. An overview of the most important sections can be seen in figure 4.3. In addition to the `text`, `.data` and `.bss` sections, the stack and the enclave SSA frame are present in the upper part of the enclave. More information about the meaning of these sections can be fount in the background chapter. (§ 2.1.6).

4.2 Development process of the bug oracle

In this section, the development process of the bug oracle will be explained. The goal of the fuzzer is to recognize bugs using a bug oracle. This bug oracle can be constructed, as explained in the previous section, by combining page faults with page table entry access and dirty bits. Page faults have a course space granularity, namely 4K. When we combine the dirty and access bits from the page table entries, information about pages which are not faulting at that time, as the type of page access can also be incorporated into the memory footprint. But before we get to that point, we take a look at the page faults themselves.

Experimental setup The experiments in this thesis are performed on a Lenovo ThinkPad T570 laptop using Arch Linux with Linux kernel `5.0.12`, Intel SGX SDK version `2.1.1` and gcc version `8.3.0`.

Enclaves used examples To conduct the tests in this thesis, multiple example enclaves are used. Part of these examples originate from a lecture given on the Space 2018 conference [28]. The code of these enclaves can be found in Appendix A.

4.2.1 Registering page faults

In this first experiment, we will try to construct a first, elementary memory footprint. Initially, every page in the address space from the enclave is made inaccessible. When a page fault happens, we allow access to the requested page, and remove access from a previous page. This keeps our working set small. Using this technique on a dummy enclave with a working window of 4 pages gives us the memory footprint in figure 4.4. (In the next part we explore what the optimal working window length is.) In this figure, we see the enclave page faults plotted chronologically. The x-axis denotes the amount of page faults that have happened since the start of execution. The y-axis shows the Enclave Page Number: the page number counted from the start address of the enclave. Access and dirty bits are not yet displayed, as the memory footprint is gradually build up over this chapter.

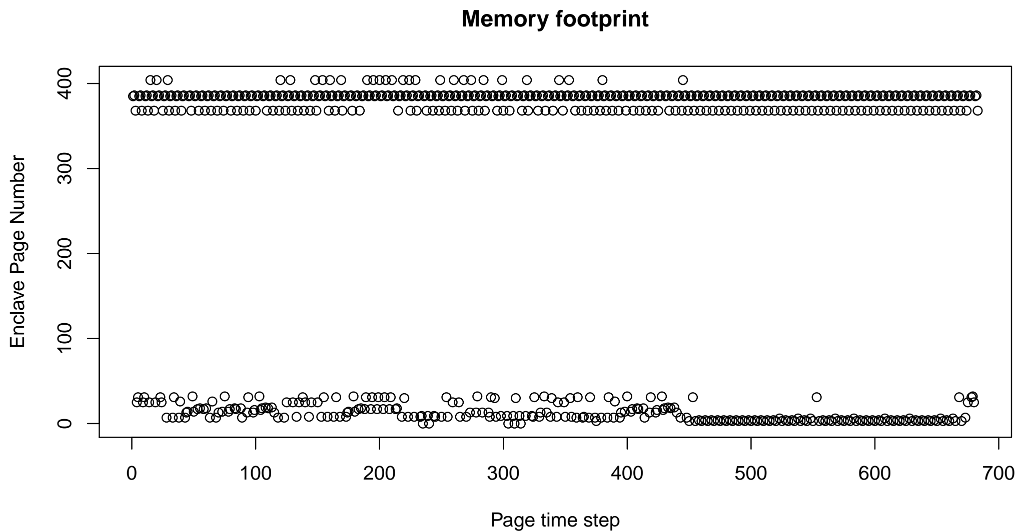


Figure 4.4: Page faults of a running enclave with a working set of 4 pages. The x-axis denotes the amount of page faults since the start of the enclave execution, the y-axis shows the Enclave Page Number of the page that faulted.

4.2.2 Adding a dynamic working window

It is not immediately clear what the size of the required minimum working window has to be to run an enclave. Choosing a working set which is too small results in an infinite loop, due to not every page needed being available. (a stack page, code page, data page and the SSA frame) Working with a dynamic working window should solve this problem. To update the current method with a dynamic working window, we will need to be able to detect loops. In general, when the enclave execution is stalling, a repeated pattern emerges: the same pages will fault over and over again. It is possible to detect this situation by keeping a counter and increasing it as long as no new pages are detected. If a new page which was not in the current working set is detected, the counter is reset. When the counter reaches a certain number, the working window is increased by one and the counter reset. An upper limit of 100 is chosen to be an initial limit for this counter.

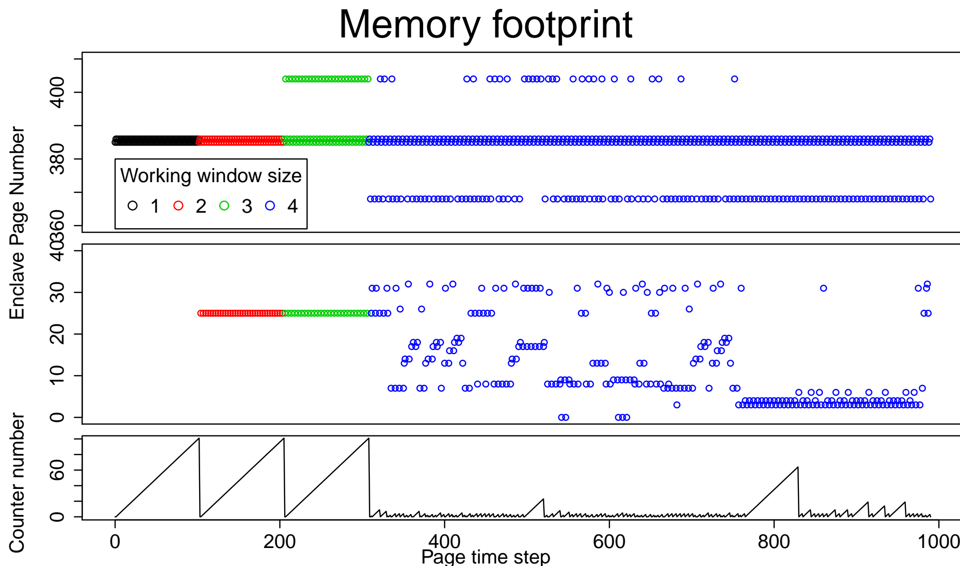


Figure 4.5: Enclave execution making use of a dynamic working window. Starting with a size of size one, the working window is gradually increased until the enclave execution is able to continue. This happens when the working window size is 4.

Monitoring the enclave using this method results in figure 4.5. The circles again represent page faults, and this time the working window size is represented as the color of the circle. The enclave memory footprint graph is split in 2 sections, showing the upper and the lower part of the enclave memory space. This is to prevent the large empty space as seen in figure 4.4. The algorithm starts with a working window size of 1 page. The black line in the lower plot shows the value of the counter discussed above. The working window gradually builds up to size 4. Around the 800th page fault, the enclave looks to be processing some data, resulting in a repeated pattern. The increasing of the counter also shows this. The enclave is not stuck in this position, but does repeatedly access the same page sequence before continuing.

To prevent raising the working window in this situation, a sufficiently high limit for the counter has to be chosen. Experimentation shows an upper limit of 100 is sufficient.

Lowering the window working size Before the access and dirty bits will combined in the detection tool, we first try to lower the working window size to a possibly optimal size. An optimal size for the working window would be the smallest size for which the enclave execution does not get stuck. Choosing a smaller working window allows for more page faults and thus more measuring points to use for the access and dirty bits. If the enclave only needs the extra pages for a small part of execution, it should be possible to lower the working window and still resume execution. To do this, we need to detect if progress is being made. Progress can simply be detected by new pages being faulted. We build a scenario where the working window is reduced after a few successful new pages are faulted. The expected result would be that the working window rises until the enclave can start or resume execution. After it is determined that progress is made by noticing new pages appearing in the working set, the working window is reduced and the enclave resumes execution with a smaller working window until it gets stuck again. This method will only be worthwhile if there are parts of the enclave that are able to run with a smaller working window. After running a test of this algorithm on a dummy enclave, the enclave seems to mostly run with a working set of 4 and only intermittently progresses with a working window size of 3. The small improvement of a short execution with a window size of 3 comes with the cost of one hundred (the counter cutoff) page faults. Because this proves to be a lot of overhead for the detection tool, the working window is chosen to be permanently 4. Although some page faults are lost when using this larger window, using the access and dirty bits can still give insight on access patterns of pages in the working set.

4.2.3 Investigating the maximal working window size

Normally, an executing program only needs access to at most 3 pages. [27] This upper limit originates from the maximum locations any instruction from the x86 architecture can access: the code page with the instruction itself and 2 data pages. During execution of Enclaves, it looks like this upper limit is 4. Investigating the cause of this extra page, brings us to the SGX SSA frame discussed in § 2.3.1. The SGX SSA frame is required to successfully execute the `ERESUME` instruction. Using the SGX-Step framework [29], we can locate the SGX SSA frame in a debugging enclave and confirm the SSA frame is the cause of the extra needed page.

Enclave memory layout Digging a bit deeper into this topic, we can also take a look at the access of the code and data sections of the enclave. Overlapping the memory layout from figure 4.3 on the memory footprint results in figure 4.6. It is clearly visible that the enclave execution accesses the `.text` section where the code resides, the `.data`, `.rodata` and `.bss` section where the data is mapped. Finally

at the top of the memory layout, the stack and the SGX SSA frame are frequently accessed.

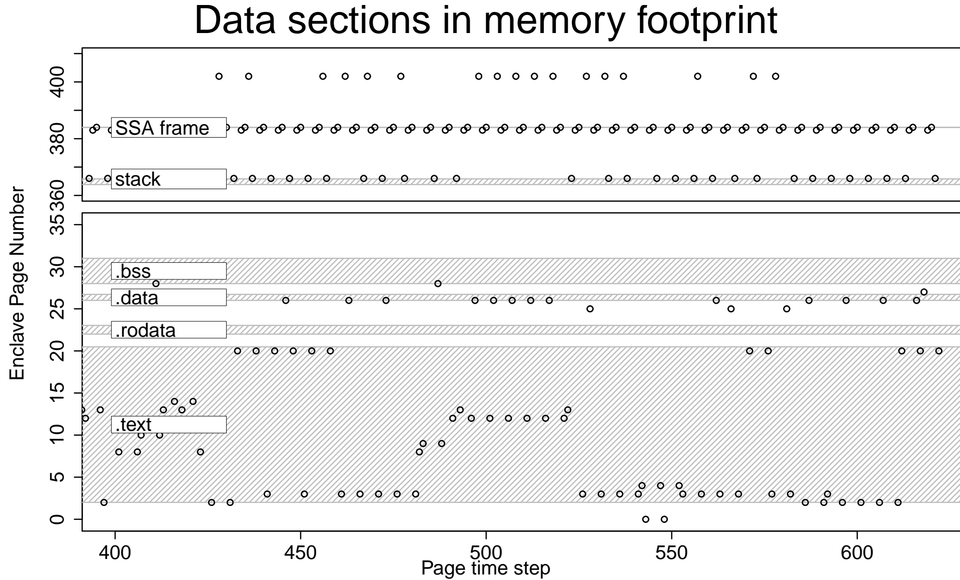


Figure 4.6: Mapping the memory layout on the enclave shows the different sections being access during enclave execution. The `.text` section contains the code of the enclave, while the `.bss`, `.data` and `.rodata` sections contain enclave data.

4.2.4 Cross-referencing the results with enclave debug data

Using the SGX-Step framework [29], it is possible to retrieve data from inside the enclave we would otherwise not have access too. One of the features of this framework is retrieving the registers from inside the enclave. We can use this information to cross check the page faults graphs with the actual stack and instruction pointers. First, we will check the instruction pointer (`rip`). Every page fault, we check the value of the instruction pointer, and add the value to the graph of the memory footprint. The result can be seen in figure 4.7. The instruction pointer is drawn as the plus signs. It is clear from the graph that the instruction pointer resides in the code pages that are currently in the working set. While other pages fault, we can also see the instruction pointer moving inside the page. It is important to notice that this is information that is not available during normal, non-debug execution of the enclave. We can do the same experiment with the stack pointer (`rsp`). An experiment confirms the location of the stack pointer and that it is included in the upper part of the memory footprint. The location of the stack is also shown on figure 4.6.

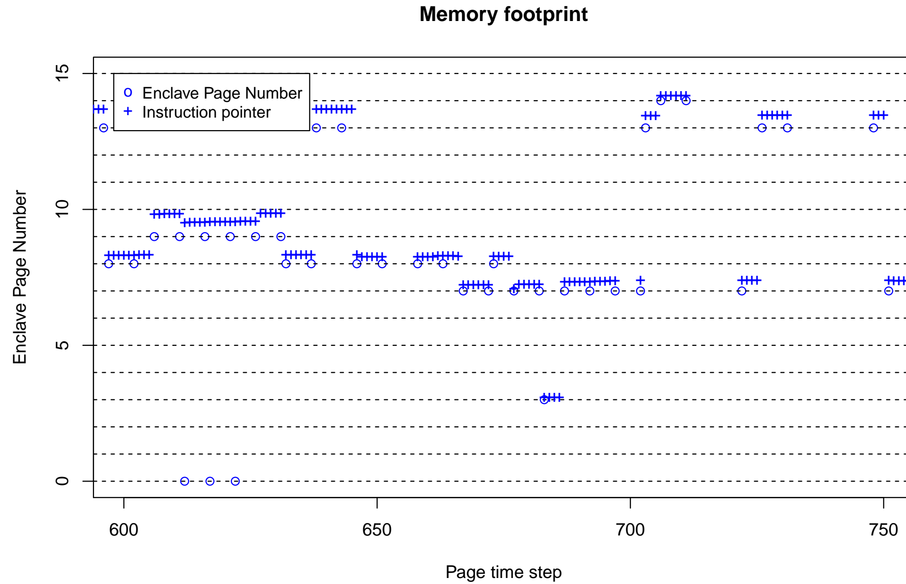


Figure 4.7: Cross checking the instruction pointer with the generated memory footprint. For clarity, only a part of the memory footprint graph is shown. The horizontal lines represent the page boundaries. The instruction pointer clearly stays within the pages that are present in the working set, confirming the determinism of the page fault oracle.

4.2.5 Comparing enclave runs

To gain more information from these page fault graphs, we need to know how deterministic the execution of an enclave really is. Depending on the code the Intel SGX SDK generates, it is possible that multiple runs of the same enclave produce different memory footprints. If multiple runs of the same enclave function result in different page fault patterns, the data is less useful in detecting possible vulnerabilities. For this experiment, we use the RSA enclave used in a lecture on Intel SGX. [28] (See § 5.2.1) We create the enclave and run one of the enclaves ECALL methods multiple times and record the page faults. The secrets and the function input remain constant. The resulting graph can be found in figure 4.8. The memory footprint clearly shows that the first time the enclave is run, additional enclave initialization code is executed. After this, the execution pattern does not change in subsequent runs. This means, at least for this example, that these page fault graphs can be used to compare multiple runs of an enclave function.

4.2.6 Adding access and dirty bits to the bug oracle

So far we have explored memory footprints made only from page faults. These graphs give us insight on how enclaves run, but only with a space granularity of 4096 bytes. This granularity also directly influences the amount of page faults that happen. If the

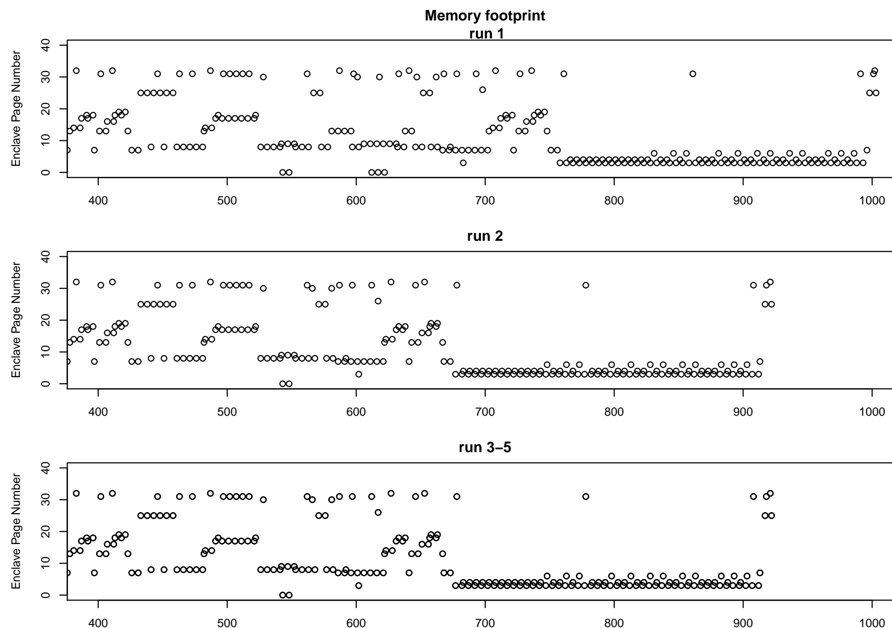


Figure 4.8: Multiple enclave runs are compared. Only the first enclave run shows additional code being executed, while further enclave executions do not change their execution patterns.

code executing stays within the same page, less new page faults happen. The access and dirty bits from the page table entries give us insight to what is happening to pages in the working set which are not faulting at that time. We can detect writes or accesses to pages using these bits in the page table entry. To test this, an experiment with a new example enclave is set up. We expect to see some pages which are only accessed and some pages which are accessed and dirty. Sometimes, when a page is not immediately used, we could see this represented on the memory footprint graph as a temporary not accessed and not dirty page. We use a different example enclave here, namely an example that executes the following function:

```
void ecall_inc_secret_maccess(int s) {
    if (s)
        a++;

    /* always access 'a', independent of the secret */
    volatile int b = a;
}
```

Depending on if a correct secret is supplied, the enclave increments a variable inside the enclave. To prevent normal page faults detecting the access to the page where the variable is located, the enclave accesses the variable every time. In this experiment, the value of `s` is set to 0: the variable `a` is accessed but not written to.

Performing the experiment results in figure 4.9. This graph combines page faults

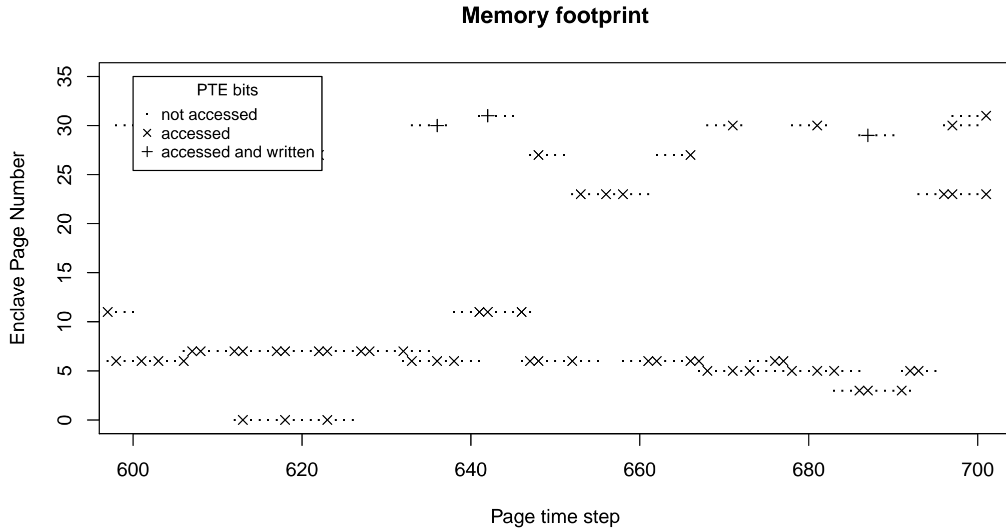


Figure 4.9: Adding the dirty and access bits to the enclave execution graph improves the details on the memory footprint graph. Page accesses and writes are now differentiable, as well as the more correct time step on which these accesses happen.

and access and dirty bits. We can clearly see which pages are only accessed and which pages are also written to. In the top left we can see a page that is accessed but not written to. Using this information, we have a more detailed representation of page accesses. When selecting the optimal window size, the decision was made to keep the working window size at 4. With the addition of the access and dirty bits, it is now possible to detect on which page faults the page access happens. This leads to more detailed memory footprint graphs which can be used to detect more hidden access patterns.

In theory it would be possible to include access and dirty bits for every page inside the enclave. However, pages which are not in the working set can not be accessed. For clarity in the graphs, it was chosen not to display the access and dirty bits for these pages.

4.3 Conclusion

In this chapter, we constructed and explored the possibilities of combining page faults with PTE access and dirty bits. Experimentation shows that a lot of information can be deducted about the internal execution of an enclave. While displaying the information retrieved from enclave execution this way is an important first step, detecting enclave runs which exhibit side channels or vulnerabilities is the important second step. The next chapter discusses the detection of vulnerabilities and evaluates the bug oracle on example enclaves and a real world enclave.

Chapter 5

Evaluating the Bug Oracle

As discussed in § 3.2, there are multiple elements to a fuzzing tool. From pre-processing to input generation to selecting interesting inputs for further evaluation. [18] In the previous chapter, the development of the bug oracle was discussed. Using this bug oracle, we can monitor the execution of enclaves and generate a memory footprint. With the memory footprint graphs, we are able to deduct information from within running enclave. Figure 5.1 shows a schematic representation of the current progress. In this chapter, we will look at how we can use these memory footprint graphs to detect interesting executions for further investigation and a way of automatically finding these interesting samples (triage). Finally, an evaluation of the bug oracle and methods in this chapter is performed on several enclaves and on one real world vulnerability in the Intel SDK.

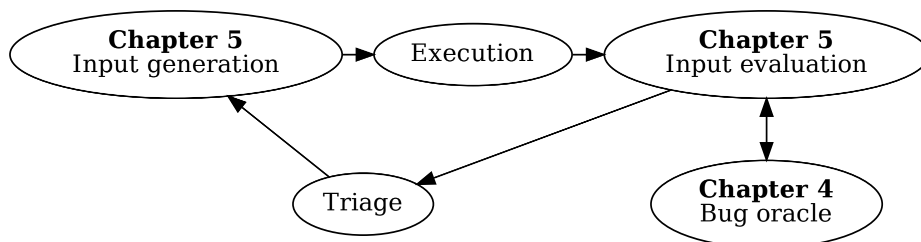


Figure 5.1: Situating the chapters of this master thesis on the general steps of a fuzzer. Chapter 4 went over the development of the bug oracle, which is used in evaluating the enclave executions. Evaluation, triage (selecting interesting inputs to continue exploring) and generation of the input are discussed in this chapter..

5.1 Detecting vulnerabilities

So far, we have implemented both page faults and access and dirty bits into the memory footprint graphs. While displaying the information retrieved from enclave execution this way is an important first step, detecting enclave runs which exhibit indications of side channels or vulnerabilities is the important second step.

5.1.1 Detectable vulnerabilities

As discussed in § 2.2, there are multiple classes of vulnerabilities. Memory vulnerabilities, Side channel attacks and speculative execution. The bug oracle uses a series of page faults and page table entry bit recordings in its memory footprint. From the three categories, speculative execution is not discussed in this thesis in relation to the bug oracle. However, the other two categories do have vulnerabilities that are visible on page level.

Memory vulnerabilities Memory vulnerabilities can be split in two subcategories: spatial and temporal [18]. Detecting these vulnerabilities is possible in multiple ways: vulnerabilities that crash the enclave are easily observed, but vulnerabilities that do not crash the enclave require a more detailed approach. In this work, the spatial type of memory vulnerabilities will be investigated. Malformed input leading to accesses over the whole enclave memory is an indicator of unwanted behaviour. Temporal vulnerabilities like use-after-free are not investigated, but can be considered future work. Schwarz et al. [24] show that it is possible to detect this category of bugs using side channels.

Side channel vulnerabilities Side channel vulnerabilities are defined as unintended information leakage due to the implementation of the system an algorithm or process is running on, and not due to a weakness in the algorithm or process itself. [37] These lead to attackers being able to deduce information about secrets from programs and devices. The bug oracle is a powerful tool to detect side channel vulnerabilities in the memory access domain. Changing memory footprints of an enclave due to changing the enclave secrets can be an indicator that a side channel vulnerability is present. By using the resulting page graphs, it may be possible to derive enclave secrets. The main challenge for this oracle is to find interesting inputs for further investigation.

API level vulnerabilities In some cases, simple mistakes or oversights by the developer can be still present in the enclave and compromise the enclave contents. Examples include forgotten debug print statements that print the secret key or a forgotten API method that can leak information. These methods can be encountered during fuzzing and detected using the framework.

5.1.2 Types of memory footprint distortions

The vulnerabilities described in the previous section are in most cases visible on the enclave memory footprint. These footprints can differ in multiple ways, where the 2 most important ones are spatial and temporal distortion.

Temporal distortion Temporal distortion happens when a different code path is taken inside the enclave, or if a based data structure leads to more code execution. The execution of the enclave can be divided in multiple blocks, representing the executed functions or blocks of code. When a function runs multiple times and page faults are generated, a repeated pattern will be visible. The left part of figure 5.2 shows a schematic example. Depending on the enclave, this repeated pattern can be dependent on the input, or on the enclave secrets. The latter case is interesting, as this differing execution path could lead to enclave secrets being exposed via the bug oracle. Where fuzzers that have access to the source or binary of the programs are able to add instrumentation, that is not possible with the SGX attacker model in mind.

Spatial distortion A spatial, or vertical movement in the graph between different enclave runs is possible when, depending on secrets or input data, other data or functions are accessed inside the enclave. This becomes only visible when the data is on different pages. This kind of movement also occurs when non intended pages in the enclave are accessed, hinting to a possibly present vulnerability. The right part of figure 5.2 shows a schematic example.

Dirty and access bits Dirty and access bits do not change the memory footprint graph in the same way the previous two causes do: they do not change based on execution flow or which data is being accessed. Dirty and access bits change based on the type of access to the memory pages of the enclave. These changing access patterns can be an indication of side channel vulnerabilities being present and are interesting to monitor, but ultimately this information makes very localized changes to the memory footprint graph, which makes them easily detectable when comparing memory footprint graphs.

In real enclave execution, memory footprint graphs will change with a combination of these spatial and temporal causes. That makes comparing multiple enclave executions difficult. However, when we only change a small part of the starting conditions, the enclave runs become more deterministic. This makes simple difference measure techniques possible.

5.1.3 Measuring differences

To automate the determination of the amount of distortions in the memory footprint graph, there are multiple mathematical methods that can help. The goal of a

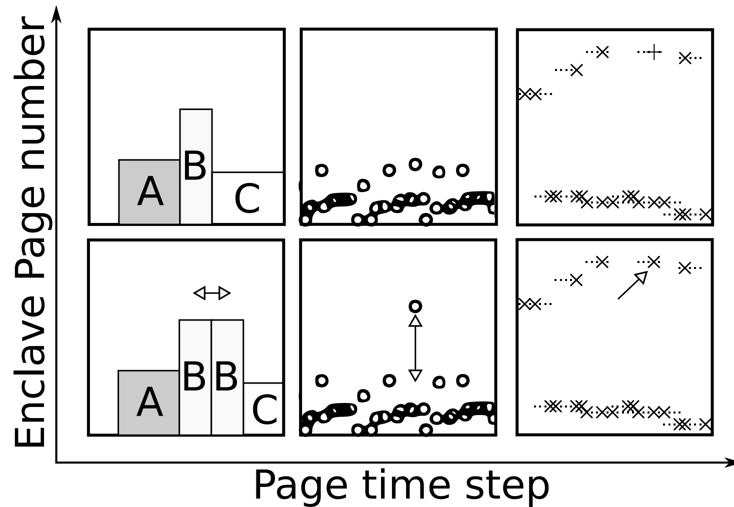


Figure 5.2: Enclave memory footprints can change in multiple ways. One possibility a change in the code execution flow (left), an other possibility is a spatial access difference (middle). A third change can be caused by a differing mode of page access, visible by monitoring the dirty and access bits (right). Enclave executions with different inputs distort the memory footprint graph with a combination of these three.

difference measure is to detect changes between enclave runs, but ultimately allow for some noise of non deterministic enclaves. Tree difference measures will be discussed:

Absolute difference The most simple form of difference metric between multiple enclave runs is just taking the absolute difference between the different enclave page numbers on a certain page fault. This way, it is possible to detect where the enclave execution starts to divert or if there are spatial inconsistencies in the enclave execution. Although this metric makes it difficult to compare enclave runs with changing input that execute different code based on the input, it can be used when the input is constant but only the enclave secrets are rotated. The power of this difference metric can be seen in the RSA enclave example (§ 5.2.1).

Edit distance A possibly better alternative to the absolute difference is the *Edit Distance*. The edit distance is a means to measure the distance between two strings by counting the minimal number of edits needed to reach one string into the other string. [32]. This is a more intuitive algorithm which could work better in detecting different code branches.

Time series analysis Comparing different time series is not a new concept. A lot of research has been done on the topic, especially in the machine learning domain. Some methods are directly applicable to the use case here. One of these methods is *Dynamic Time Warping*. Dynamic time warping is a pattern recognition algorithm

that can be used to measure the similarity between two time series. Using the similarity between memory footprint graphs allows to ignore differing execution patterns in the beginning of execution and also focus on the later components of the memory footprint graph. Because of its complexity, this difference metric was not implemented.

5.1.4 Enclave input space

Before interesting memory footprints generated by the bug oracle can be analysed, input for the enclave has to be generated. Instead of only providing random data to the enclave, we can take a smarter approach. Looking at the attacker model, Intel SGX prevents the host from accessing the enclave memory. Because the attacker initially has absolutely no access to the memory, finding a vulnerability which allows to access the complete enclave memory is very powerful. To aid in the discovery of these kind of enclave memory accesses, we can supply the enclave with smarter input.

Enclave parameters and state There are two main parts to providing input to the enclave. On one hand, there are the parameters of the ECALLs and OCALLs of the enclave. These parameters are used as input for the functions inside the enclave and are an important part of input generation. Parameters are completely controlled by the attacker and thus a powerful tool for fuzzing. On the other hand, there is the state of the enclave. The state encompasses the internal state of an enclave. This state consists of internal variables, secrets and more. Every time the enclave is initialized and loaded into memory, the state of the enclave is the same due to the attestation principle of Intel SGX, but the state can change by executing the functions of the enclave. An attacker does not have any direct control to the state of an enclave, as it generally changes as a side effect of executing enclave functions. Often, the state is an important part of directing the execution flow of the enclave. An example of the state being leaked from the enclave could be seen in figure 4.8 in the previous chapter. The extra code being executed leaks the value of an internal variable `g_enclave_state`, a variable used to keep track of the initialization state of the enclave. Changing the state of the enclave can help to uncover possible side channels and vulnerabilities of the enclave.

Smart parameter input As a way to more efficiently fuzz an enclave, a smarter way of generating the parameter input can be devised. One example can be supplying pointers pointing to the inside of the enclave. If the enclave forgets to check the location of the pointers, or it has a buffer overflow leading to the code jumping to a certain location, this behaviour will be visible in the enclave memory footprint. This is the primary technique used when evaluating the enclaves in this chapter.

Smart state input Another way of choosing an interesting input is changing the state of the enclave. Instead of keeping the secrets stationary, we can reinitialize the enclave again and again and compare the different enclave runs. A changing secret

can than be an indicator of a side channel. This is further explained in § 5.1.4. It is however not always possible to change the secrets and the state of an enclave at will. It is for example possible that a vendor supplies specific unique secrets to an enclave after remote attestation. But when it is possible to change the secrets and state of an enclave in a controlled manner, it can be a tool in detecting important side channel vulnerabilities.

Apart from supplying pointers to the inside of the enclave as parameters, certain pointers to the outside of the enclave can also be used. If these pointers point to data structures under control of the attacker, they can also be made inaccessible and be part of the enclave memory footprint. Enclave data can still be leaked to these outside structures in cases like failing bound checks. This is an interesting feature which is part of possible future work.

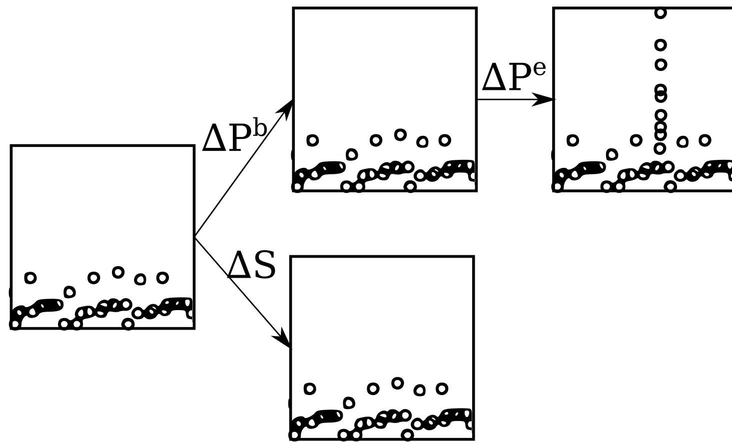


Figure 5.3: The multiple steps in detecting interesting enclave behaviour. First, a normal static enclave memory footprint should be made by combining a static execution of the enclave with constant parameter input and state, and enclave execution with benign changing parameter input (ΔP^b). Changing the enclave state (ΔS) or choosing specially crafted parameter input (ΔP^e) can lead to discovering vulnerabilities or possible side channel vulnerabilities.

Finding interesting enclave executions

When comparing multiple enclave executions, supplying random input to the functions can already expose interesting behaviour. Random enclave memory access and crashes can be detected without knowing how normal execution of the enclave looks like. Simple comparisons do have trouble with detecting side channels or more subtle differences. For a more in-depth analysis, a more complex approach is needed. With this simple comparison, even in non deterministic enclaves, patterns like access over the entire enclave emerge.

Complex enclave execution comparison To make better use of the bug oracle and memory footprints, we can take a step further from comparing only enclave runs with random parameter inputs. A more complex approach for comparing multiple runs of an enclave is schematically represented in figure 5.3.

First, we need to prepare a base execution footprint with a normal execution: the enclave is run multiple times with the same parameter input and the same state. This gives gives a first indication of the determinism of the enclave. Generates every enclave execution the same memory footprint, or are there differences? Further exploring the parameter input differences, the next step is generating different benign parameter inputs (ΔP^b) and executing the enclave on those inputs while keeping the state constant. This again gives us insight in the determinism of the enclave code: the parts that change even without changing the parameters or state and the parts of the memory footprint that change when the parameters change. The memory footprint we have at this point can be used as a basic memory footprint when exploring attacker input to the parameters in the final step (ΔP^e). Another possible step is the executing the initial enclave with constant parameter input, but (if this is an option) changing or renewing the enclave secret and state. If the execution flow of the enclave differs from the previous executions, this may point to a possible side channel vulnerability.

Now is the time to execute the enclave and supplying our own random or specifically chosen input. The memory footprint graphs generated in this step can be compared to the base memory footprint from the previous step. The choice of difference measure is important in this step, as it can reveal more information. New code coverage or possibly interesting behaviour can be further explored.

5.2 Testing on example enclaves

Keeping the evaluation of interesting enclave executions in mind, we will now evaluate the bug oracle against multiple vulnerable enclaves. These evaluations will show the ability of this bug oracle to detect vulnerabilities.

5.2.1 Space 2018 RSA enclave

The fist evaluation will be done on the RSA enclave toy enclave from the Space 18 conference tutorial. RSA is an encryption algorithm based on the difficulty of factorization. Two prime numbers, p and q, are chosen and used to generate public key e, a private key d and n, the product of p and q. From these, e and n are made public and used as encryption key. A message M is encrypted into cipher text C by calculating $C = M^e \mod n$. Decryption of C can be done using the private key d by calculating $M = C^d \mod n$. [23].

In this enclave, the RSA algorithm is implemented using the *square-and-multiply* algorithm. The calculation of $M = C^d \mod n$ is done by repeatedly squaring m each iteration. If on an iteration i, the ith bit of d is set, m is multiplied by c. This algorithm is implemented in the function `modpow`:

```
/*
 * Computes a^b mod n.
 * (long long multiplication prevents overflow)
 */
int modpow(long long a, long long b, long long n) {
    long long res = 1;
    uint16_t mask = 0x8000;

    for (int i=15; i >= 0; i--) {
        res = square(res, n);
        if (b & mask)
            res = multiply(res, a, n);
        mask = mask >> 1;
    }
    return res;
}
```

The function `modpow` uses 2 helper functions, `square` and `multiply`. These functions respectively square and multiply 2 large numbers. It is a common occurrence for cryptographic libraries to have these functions defined to multiply or square large numbers. In this enclave, the `square` and `multiply` functions are defined on different pages:

```
/* See asm.S */
uint64_t square(uint64_t a, uint64_t n);
uint64_t multiply(uint64_t a, uint64_t b, uint64_t n);
```

As counter measure against some side channel attacks, a method known as *message blinding* is used in this enclave. This method first multiplies the message with a randomly chosen factor r, encrypts/decrypts the message and then removes the factor r by multiplying with the inverse of r. This message blinding prevents side channel attacks where a specially crafted message can differentiate between the `square` and `multiply` functions. [33]

The decryption function of the enclave looks like this:

```
int rsa_n = 57677;
int rsa_e = 11;
int rsa_d = 20771; //16'b0101000100100011

int ecall_rsa_decode(int cipher) {
    int i, mask, r = 0;
    long long res;

    /* Blinding with 16-bit random factor. */
    if (sgx_read_rand((unsigned char*) &r, 2) != SGX_SUCCESS)
        return 0;

    cipher = cipher * modpow(r, rsa_e, rsa_n);

    /* Decrypt blinded message with square and multiply algorithm. */
    res = modpow(cipher, rsa_d, rsa_n);
```

```
    /* Unblind result. */
    return (res * inverse(r, rsa_n)) % rsa_n;
}
```

The function `sgx_read_rand` is a SGX helper function to safely generate random numbers. The function `inverse` calculates the inverse of an integer. Finally, the enclave has a method to reset the secret. This method can be used to automatically generate a new secret inside the enclave:

```
int ecall_reset_secret() {
    int r = 0;

    if (sgx_read_rand((unsigned char*) &r, 2) != SGX_SUCCESS)
        return 0;

    rsa_d = r;
    return 1;
}
```

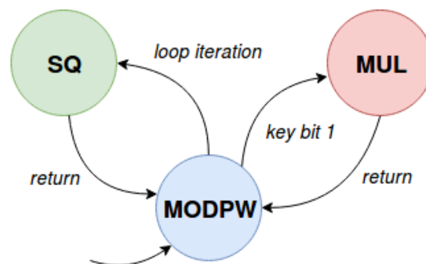


Figure 5.4: The state machine of the `modpow` function in the RSA enclave. This figure was taken from the space 18 tutorial repository. [28]

Vulnerability

This enclave is vulnerable to a page fault side channel attack. By monitoring the enclave memory footprint generated by the bug oracle, it is possible to detect a page fault sequence leaking the secret key.

Figure 5.4 shows the state machine of the `modpow` function. For ever bit in the key, the `square` function is executed. The `multiply` function however is only executed when the current bit of the key is 1. Capturing this page fault sequence effectively leaks the entire key to the memory footprint.

Detecting the side channel

To evaluate the bug oracle, we take the enclave we discussed as a black box enclave. The only known information we start with is the enclave API, its OCALLs and ECALLs. Running the enclave multiple times with the `ecall_rsa_decode` ECALL

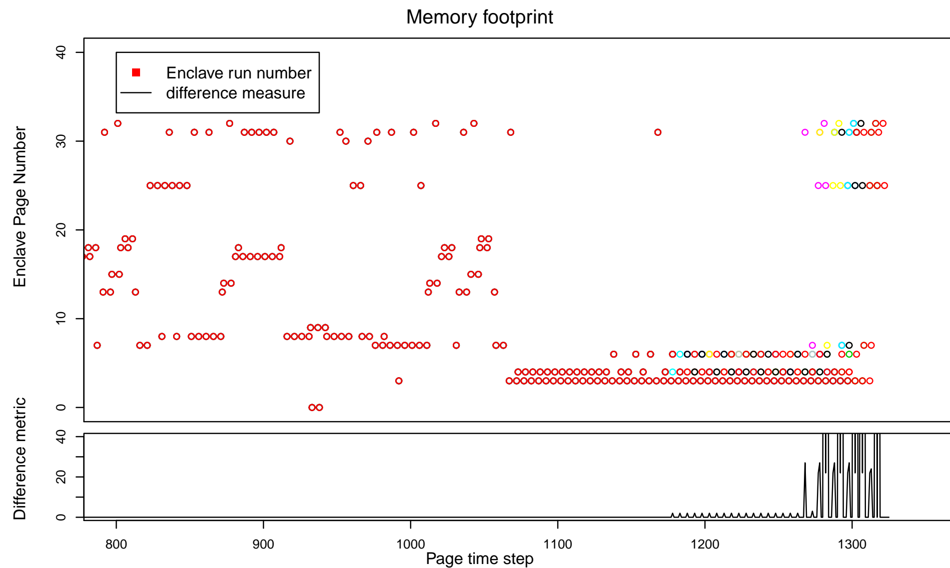


Figure 5.5: Comparing multiple runs of the RSA example enclave with a changing secret and a constant input. Each new enclave run has a different color. The memory footprint graph clearly shows a change in execution flow, which can an indication of a side channel vulnerability being present in the RSA enclave.

results in the same memory footprint every time. Even when the parameters of the decode function are changed, the memory footprint stays the same.

The next step is using the secret reinitialization ECALL we have to our disposal. Using this ECALL we can supposedly change the internal secret (the encryption key) of the enclave. Running the enclave multiple times with different secrets and the same input results in figure 5.5. Clearly, the execution flow changes depending which secret was used. The difference metric used in this example is the absolute difference metric (§ 5.1.3).

From this, it can be reasonably deducted that secret data from the enclave alters the execution flow and thus also possibly leaks the information via a side channel vulnerability. Further investigation can be undertaken to extract the secret.

5.2.2 Space 2018 increment secret enclave

The increment secret enclaves appears in the Space 18 tutorial on SGX side-channel tutorial as an example of writing idempotent code that does not change based on the secret or input values. The example was shortly used in § 4.2.6. There is only one function inside the enclave:

```
void ecall_inc_secret_maccess(int s)
{
    if (s)
        a++;
```

```
    /* always access 'a', independent of the secret */
    volatile int b = a;
}
```

The idea is that following a check on a variable `s`, a value is written to variable `a`, which is located on a different page. This write to the page where `a` resides (P_a), is visible as a page fault on the memory footprint graph. As a defensive idea, `a` is always accessed, independent of the previous code. Running this enclave with both `s = 1` and `s = 0` results in the same page fault sequences.

Vulnerability

The problem in this enclave is that the access to the page where `a` resides can still be seen in the access and dirty bits of the enclave. When variable `a` is incremented, the dirty flag on P_a page is set. When no increment happens, `a` is only accessed and therefor the dirty bit will not be set. This difference is visible in the memory footprint graph.

Detection

Executing this enclave with the fuzzer in a black box scenario, no noticeable differences can be seen. The enclave always executes with the same page fault sequence (except the first, initial enclave execution). As the enclave ECALL only accepts an integer, random inputs to the system do not cause any different patterns. However, when looking at the access and dirty bits, a change emerges. When the value `s=0` is supplied to the enclave, a page does not get written to, but with a value `s>1`, the dirty bit for the page gets set. The result can be seen in figure 5.6. The difference appears in the upper right of the graph.

5.3 Real world application

March 2018, the Intel published a revision of their Intel SGX SDK. The reason for this update was an issue with the `edger8r` tool present in the SDK, found by KU Leuven researchers.

5.3.1 edger8r

Edger8r is a tool in the Intel SDK which generates the harness around enclave code. It handles the definition and generation of ECALLS and OCALLS, as well as some input validation for these functions. Every enclave written with the Intel SGX SDK uses edger8r to build the interface routines to the enclave. Normally, a developer does not interact with the generated code. This combination makes mistakes in edger8r even more dangerous.

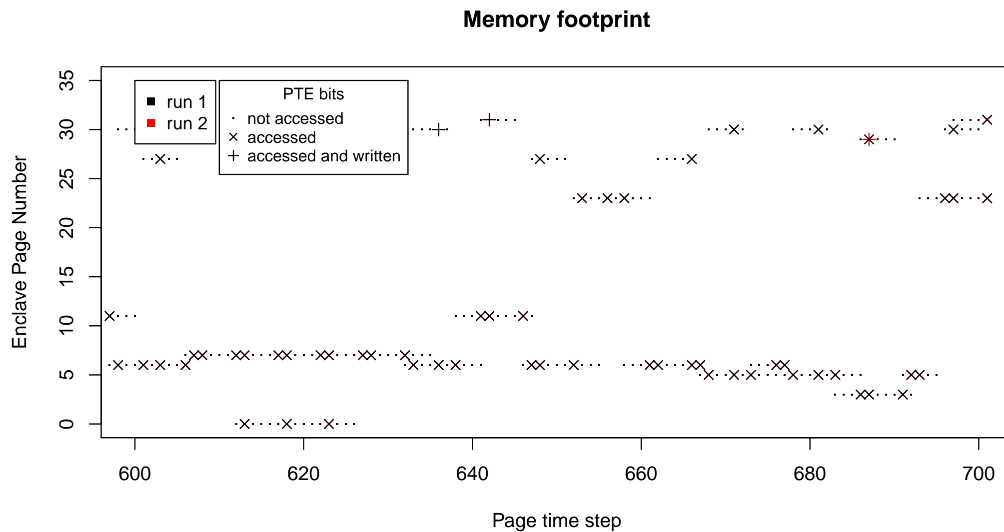


Figure 5.6: The upper right of the memory footprint graph shows how in one enclave run, a page is accessed but not modified, and in the other run the same page is only accessed. By monitoring these changes, it is possible to find potential leaks of enclave secrets.

Finding the vulnerable version The vulnerability exists in and before Intel SGX SDK version 2.1.1, and was fixed in an update to Intel SGX SDK version 2.1.102. [15] An older, vulnerable version of edger8r was available via the Linux SDK git repository at the tag `sgx_2.1.1`. Edger8r is a tool written in the Ocaml language, which since made some deprecating changes to packages used in the older version of the edger8r codebase. Instead of finding the correct versions of the Ocaml packages, small fixes in the edger8r code base were applied to get things working again. No main functionality was changed.

5.3.2 Vulnerability

In contrast to the previous enclaves, the problems in this real world application are due to a mistake in one of the build tools. During the development of enclaves, the developer needs to declare the ECALLs and OCALLs in a separate `.edl` file. Together with defining the edge calls, the arguments of the function have to be annotated with attributes. (§ 2.3.1) One of these attributes is the `string` attribute. This attribute denotes a character array with a null terminated string. Edger8r uses this information to generate an edge routine which calculates the string length with `strlen`. This function checks the length of a string by going over the string array until a null byte is encountered.

Part of this edge routine lies inside the enclave, and uses `strlen` on the array before checking if the pointer is located outside the enclave. Normally, when a string

to the outside of the enclave is supplied, the check succeeds without problems. But when a pointer to the inside of the enclave is supplied, the enclave will execute `strlen` on memory inside the enclave. As the enclave has access to the memory inside the enclave, `strlen` will execute until it encounters a null byte. This effectively leaks memory from the enclave. Using other techniques, this vulnerability can then lead to severe memory leaks. The code generated by this version of edger8r can be found in the Intel vulnerability report. [15]

For this experiment, a simple enclave with a single ECALL was used. The `.edl` file has the following contents:

```
enclave {
    trusted {
        public void ecall_pointer_string([in, string] char *str);
    };

    untrusted {};
};
```

The method `ecall_pointer_string` is a dummy function and looks like this:

```
void ecall_pointer_string(char *str) {
    volatile int b = 0;
}
```

This is enough for the vulnerability to be present after compiling the enclave with the flawed edger8r version.

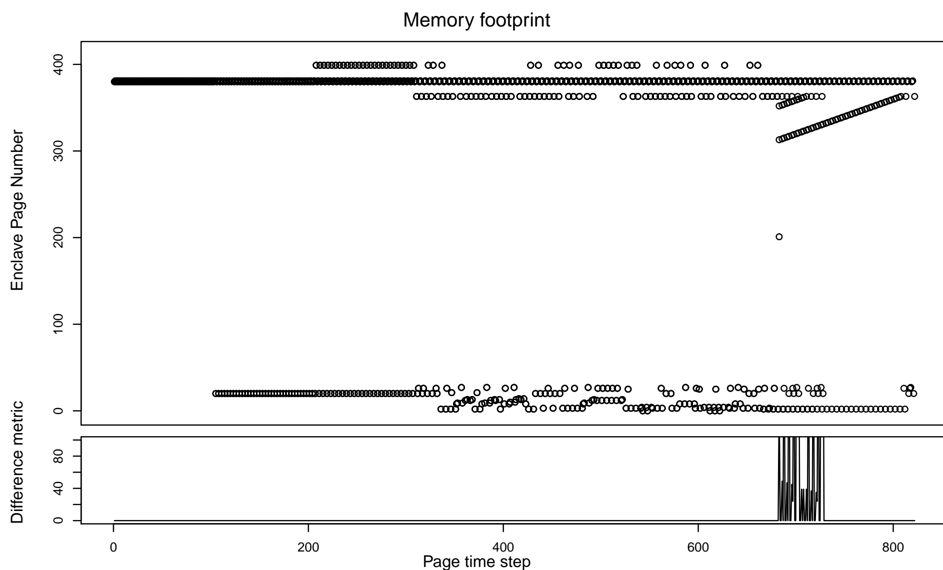


Figure 5.7: Fuzzing an enclave compiled with an older version of the Intel SGX SDK leads to this graph. Clearly, the enclave is accessing addresses in the enclave that are supplied by the attacker. This is a clear indication of a vulnerability being present.

5.3.3 Detection

When detecting this vulnerability, we use the steps previously defined. Running the enclave repeatedly in the intended way gives us a base memory footprint that follows the expected deterministic view. Changing the input parameter shows no differences on the memory footprint graph. As there are no secrets to reset, no direct side channels are detectable. When we start supplying more specific inputs to the enclave, namely pointers to the inside of the enclave, only then we start seeing interesting behaviour on the memory footprint graph: the enclave accesses different pages inside the enclave. After executing the enclave multiple times, a clear line shows the point in the execution where the unchecked memory access happens. Figure 5.7 shows a few enclave runs from the fuzzing campaign. There are some conclusions that can be taken from this enclave. First, a very noticeable enclave page access in the middle of the memory graph can be seen. This is a memory access which does not occur in previous runs and points right to the page the supplied pointer pointed to. The complete graph features many of these accesses. At the top of the memory graph, a more interesting phenomenon can be seen. A memory access in the upper part of the enclave memory leads to a continued access to the next upper part of the enclave memory. Knowing that the cause of the vulnerability is a `strlen` that accepts the supplied input, no matter its location, this area of the enclave contains data without any null bytes.

5.3.4 Further exploitation

When a vulnerability like this string length vulnerability is found, it can be further exploited using other techniques like the SGX-Step framework. [29] The SGX-Step framework has the possibility of counting the amount of instructions that happen during enclave execution. This can be used in combination with the string length to map out all zero bytes in the enclave. Starting with the first address of the enclave, the string length function is called and the number of instructions is counted. The function `strlen` continues until it reaches a null byte. Counting the amount of instructions lets the attacker know where the next pointer to the enclave memory could point. Successive application of this method results in a map of every null byte within the enclave memory.

5.4 Conclusion

In this chapter, the ability to detect vulnerabilities on memory footprint graphs was discussed, concluding that although the memory footprint of an enclave can be distorted in numerous ways, vulnerabilities are detectable with a good distance measurement technique. Answering to the hypothesis posed in § 3.1.1, it is possible to elevate fuzzing of Intel SGX enclaves above black box fuzzing. Using a bug oracle combined from a page fault side channel and the PTE access bits side channel, a working method to detect certain vulnerabilities was created. Although future work

can improve much of the current implementation, the tool presented in this thesis is able to detect an existing vulnerability in the Intel SGX SDK.

Chapter 6

Conclusion

6.1 Research question

In this thesis, we tried to find an answer to the hypothesis that finding vulnerabilities in Intel SGX Enclaves by fuzzing could be enhanced over black box fuzzing by using side channel measurements. (see § 3.1.1) We went on a search for usable side channels and the vulnerabilities that could be detected using those side channels. What makes fuzzing enclaves different from fuzzing normal programs?

6.2 General conclusion

Chapter 2: Background and related work The background chapter revealed that Intel SGX leaves memory management to the untrusted operating system and explicitly does not include side channels in the threat model. This opens the door to using side channels as a means of monitoring enclaves. Previous work shows that exploitation of vulnerabilities within enclaves is a well researched topic, but the detection and availability of vulnerabilities is needed to execute these attacks.

Chapter 3: Fuzzing Intel SGX enclave execution In the following chapter, it becomes clear that fuzzing Intel SGX enclaves differs from the normal white-box fuzzing, but does not completely fall into the black box fuzzing category. By using side channels, a fuzzer can use that information to operate. To allow a tool to make use of an available side channel, a bug oracle needs to be developed. The difference in fuzzing enclaves and applications running in a normal mode is discussed, with the conclusion that enclaves, due to their isolation are not able to be instrumented properly, but also can behave more deterministically and more noise free.

Chapter 4: Developing a Bug Oracle Chapter 4 starts with the desired properties of a bug oracle: determinism, granularity and efficiency. The bug oracle used in this work is the combination of page faults and page table bits. Due to the deterministic nature of page faults, this bug oracle is very deterministic compared to for example cache based enclaves. A trade-off when using page faults is the limited

spatial granularity of 4KB and a non standard temporal granularity, due to the time between page faults being dependent on the data accesses and executing code within the enclave.

The importance of keeping he working set small is experimentally demonstrated, concluding with an optimal working window size of 4. Combining access and dirty bits into the memory footprint graph gives the fuzzer insight in the type of page access and a more detailed view of when a page is accessed. The determinism of simple enclaves compiled with the Intel SGX SDK is confirmed, opening the way to further develop the use of the memory footprint graph.

Chapter 5: Evaluating the Bug Oracle Chapter 5 discusses the usage of the bug oracle to detect possible vulnerabilities. First, an overview of possible detectable memory vulnerabilities using the bug oracle is discussed, from which spacial memory safety vulnerabilities and side channel vulnerabilities are chosen for further analysis.

These vulnerabilities can distort the memory footprint graph in both temporal and spatial dimensions. Multiple difference measures to detect these distortions are presented, from which the edit distance is chosen for further evaluation. This due to the simplicity compared to the implementation of time series analysis and edit distance.

An enclave execution is defined by both the input parameters and the internal enclave state. By noticing the difference and exploring the input space of the enclave, it is possible to rigorously investigate the influence of parameters and state on the memory footprint graph. Choosing smart input for the enclave state space can result in better detection rates. For the parameter input, pointers to the inside of the enclave are interesting, while for the enclave state, changing the secrets leads to interesting observations.

The final part of this thesis is the evaluation of the bug oracle on enclave examples. The oracle is able to clearly detect a side channel present in a RSA algorithm. The bug oracle is also able to detect a conditional access of a variable using the difference between a page access and a page write in the 'increment secret' example. Ultimately, the bug oracle is tested on a real world enclave. A side channel vulnerability generated in the edge routines by the edger8r tool from the Intel SGX SDK was successfully detected, showing the real world value of this tool.

6.3 Answering the research question

It is clear that we have found a method to measure enclave execution and use this information to guide a fuzzer. By combining the page fault side channel with the dirty and access bits side channel, a bug oracle was created with which it is possible to apply a grey box fuzzing strategy to Intel SGX enclaves. We are able to detect multiple classes of vulnerabilities, including memory safety vulnerabilities and side channel vulnerabilities. Furthermore, because the Intel SGX implementation, enclaves provide a stable and relatively noise free environment for a fuzzing target.

The automatic detection of vulnerabilities is not fully implemented, but future work can still make this a possibility.

6.4 Future work

Future work certainly could take a look at better comparing enclave runs. While the current method works for simple enclaves, more complex behaviour requires a better mapping of enclave coverage on the memory footprint graphs to the enclave runs. More types of vulnerabilities can be discussed and discovered in relation the memory footprint graph, for example the temporal memory safety vulnerabilities.

The large attack surface on the reentry points to the enclave after an OCALL has not been fully developed in this thesis. The Iago attacks [6] show that applications rarely defend against malicious returns from system calls. Since the threat model of Intel SGX assumes the operating system to be untrusted, and because of the intentional easy portability of applications to Intel SGX enclaves, detecting vulnerabilities in this setting is probably worthwhile to explore.

In this thesis, the usage of the Intel SGX SDK was assumed. There are however other SDK in development, like Microsoft's Open Enclave [19] and Baidu's Rust SGX SDK [2]. Exploring the possibility of removing the coupling of this tool with the Intel SGX SDK can be researched.

Finally, although some progress to developing an automated tool was performed, full automation has not been achieved. Enumerating the enclave ECALLs, generating a fuzzer harness and the flagging of interesting executions is left as future work.

Appendix A

Fuzzer source code

The code of the fuzzer and examples used in this thesis can be found in the following public repository: `https://github.com/mosterdt/fuzzing-intel-sgx`

Bibliography

[1] A. ARM. Security technology building a secure system using trustzone technology (white paper). *ARM Limited*, 2009.

[2] Baidu. rust-sgx-sdk. `https://github.com/baidu/rust-sgx-sdk`, 2019.

[3] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi. The guard's dilemma: Efficient code-reuse attacks against intel SGX. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1213–1227, 2018.

[4] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert. Software mitigations to hedge aes against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006:52, 2006.

[5] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD, Aug. 2018. USENIX Association.

[6] S. Checkoway and H. Shacham. Iago attacks: why the system call api is a bad untrusted rpc interface. In *ASPLOS*, volume 13, pages 253–264, 2013.

[7] T. I. S. Committee et al. Executable and linkable format (elf). *Specification, Unix System Laboratories*, 1(1):1–20, 2001.

[8] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE Symposium on Security and Privacy*, pages 45–60. IEEE, 2009.

[9] V. Costan and S. Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[10] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, 2016.

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[12] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129. IEEE, 2000.

[13] Intel. Intel® software guard extensions (intel® sgx) sdk for linux* os. *Developer's manual*, Revision 2.4, 2018.

[14] Intel. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide*, 2019.

[15] Intel. Intel software guard extensions (sgx) sw development guidance for potential edger8r generated code side channel exploits. March 2018, Revision 1.0.

[16] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539, 2017.

[17] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.

[18] V. J. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. Fuzzing: Art, science, and engineering. *arXiv preprint arXiv:1812.00140*, 2018.

[19] Microsoft. openenclave. `https://github.com/openenclave/openenclave`, 2019.

[20] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[21] J. T. Mühlberg. Postgraduate course internet-of-things: Iot security module. November 2018.

[22] T. Newsham. Format string attacks, 2000.

[23] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[24] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 587–600. ACM, 2018.

[25] M. Schwarz, S. Weiser, and D. Gruss. Practical enclave malware with intel sgx. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 177–196. Springer, 2019.

[26] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *NDSS*, 2017.

[27] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 317–328. ACM, 2016.

[28] J. Van Bulck. sgx-tutorial-space18. `https://github.com/jovanbulck/sgx-tutorial-space18`, 2018.

[29] J. Van Bulck, F. Piessens, and R. Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, page 4. ACM, 2017.

[30] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 178–195. ACM, 2018.

[31] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, 2017.

[32] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[33] M. F. Witteman, J. G. van Woudenberg, and F. Menarini. Defeating rsa multiply-always and message blinding countermeasures. In *Cryptographers' Track at the RSA Conference*, pages 77–88. Springer, 2011.

[34] R. Wojtczuk. The advanced return-into-lib (c) exploits: Pax case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, 2001.

[35] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[36] M. Zalewski. American fuzzy lop, 2019.

[37] Y. Zhou and D. Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptology ePrint Archive*, 2005:388, 2005.

Fiche masterproef

Student: Thomas De Backer

Titel: Fuzzing Intel SGX Enclaves

Nederlandse titel: Fuzzing Intel SGX Enclaves

UDC: 681.3

Korte inhoud:
Dit werk onderzoekt een manier om automatisch fouten op te sporen in Intel SGX enclaves door middel van fuzzing. Het fuzzen van Intel SGX enclaves verschilt van het normale white-box fuzzing, maar valt niet volledig onder de categorie van het black-box fuzzing. Een bug-orakel wordt ontwikkeld door twee beschikbare side channels te combineren: een page fault side channel en een side channel dat gebruikt maakt van de toegang- en schrijf-bits in de page table entries. Gebruik van het computergeheugen door een enclave wordt gemonitord met een 4KB-granulariteit. Door de toegankelijke pagina's te beheren en de werkset klein te houden, worden gedetailleerde geheugenvoetafdrukken vastgelegd en verfijnd met behulp van de invoerbits van de page table entries. Deze geheugenafdrukken kunnen vervolgens worden gebruikt om mogelijke kwetsbaarheden te detecteren. De categorieën van detecteerbare kwetsbaarheden en hun invloed op de geheugenvoetafdrukgrafieken worden besproken. Om het genereren van invoer door de fuzzer te verberen, worden de types invoer van enclaves onderzocht en wordt slimme invoer voor elk invoertype besproken, wat resulteert in betere detectiepercentages voor kwetsbaarheden. Door geheugenafdrukken met elkaar te vergelijken met behulp van een gekozen verschilstatistiek, kan de gebruiker interessante enclave-input en -output vinden en gebruiken voor verder onderzoek. Ten slotte wordt een evaluatie van de fuzzer en het bug-orakel uitgevoerd op voorbeeld enclaves. Het orakel kan side channels gemakkelijk detecteren. Uiteindelijk wordt het bug-orakel getest op een real world enclave, waarin de tool in staat was om een eerder gevonden bug in de edger8r-tool van de Intel SGX SDK te detecteren, wat de meerwaarde voor deze tool benadrukt.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Gedistribueerde systemen

Promotoren: Dr. J.T. Mühlberg
Prof. dr. ir. F. Piessens

Assessoren: Dr. A. Akkasi
Dr. R. Strackx

Begeleiders: Dr. J.T. Mühlberg
ir. J. Van Bulck